Theory Comput Syst (2014) 55:685–718
DOI 10.1007/s00224-013-9443-6

The Complexity of Compressed Membership Problems for Finite Automata

Artur Jeż

Published online: 18 January 2013

Abstract In this paper, a compressed membership problem for finite automata, both deterministic (DFAs) and non-deterministic (NFAs), with compressed transition labels is studied. The compression is represented by straight-line programs (SLPs), i.e. context-free grammars generating exactly one string. A novel technique of dealing with SLPs is employed: the SLPs are recompressed, so that substrings of the input word are encoded in SLPs labelling the transitions of the NFA (DFA) in the same way, as in the SLP representing the input text. To this end, the SLPs are *locally decompressed* and then *recompressed* in a uniform way. Furthermore, in order to reflect the recompression in the NFA, we need to modify it only a little, in particular its size stays polynomial in the input size.

Using this technique it is shown that the compressed membership for NFA with compressed labels is in NP, thus confirming the conjecture of Plandowski and Rytter (Jewels Are Forever, pp. 262–272, Springer, Berlin, 1999) and extending the partial result of Lohrey and Mathissen (in CSR, LNCS, vol. 6651, pp. 275–288, Springer, Berlin, 2011); as this problem is known to be NP-hard (in Plandowski and Rytter, Jewels Are Forever, pp. 262–272, Springer, Berlin, 1999), we settle its exact computational complexity. Moreover, the same technique applied to the compressed membership for DFA with compressed labels yields that this problem is in P, and this problem is known to be P-hard (in Markey and Schnoebelen, Inf. Process. Lett. 90(1):3–6, 2004; Beaudry et al., SIAM J. Comput. 26(1):138–152, 1997).

This work was partially supported by National Science Centre (NCN) grant SONATA number DEC-2011/01/D/ST6/07164, 2011–2014.

A. Jeż (✉)
Max Planck Institute für Informatik, 66123 Campus E1 4, Saarbrücken, Germany
e-mail: ajez@mpi-inf.mpg.de

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

Keywords Compressed membership problem · SLP · Finite automata · Algorithms for compressed data

1 Introduction

1.1 Compression and Straight-Line Programmes

Due to ever-increasing amount of data, compression methods are widely applied in order to decrease the data's size. The stored data is processed from time to time and decompressing it on each occasion is wasteful. Thus there is a large demand for algorithms working directly on the compressed representation of the data, without explicit decompression. Such task is not as hopeless as it may seem at the first sight: it is a popular outlook that compression basically extracts the hidden structure of the text and if the compression rate is high, the text must have a lot of internal structure. So if data is compressed well, it has a structure that can be exploited by algorithms. In some sense such an intuition is correct: efficient algorithms for fundamental text operations (pattern matching, checking equality, etc.) are known for various practically used compression methods (LZ, LZW, their variants etc.) [10–13, 15].

Practical compression methods, like LZW or LZ variants, differ both in main idea as well in details and so also algorithms for data compressed using various compression methods are different as well. This leads to a plethora of algorithms for various compression variants and string operations [2, 4, 7, 8, 10–13, 15–17, 22, 23, 35, 36]. However, a different approach is also explored: for some applications and for most of theory-oriented considerations it would be useful to *model* the practical compression standard by a more mathematically well-founded method. This idea lay at the foundations of the notion of *Straight-Line Programms* (SLP), whose instances can be simply seen as context-free grammars generating exactly one string. In particular, SLPs belong to a broader family of *grammar-based compression methods*.

SLPs are the most popular theoretical model of compression. This is on one hand motivated by a simple, 'clean' and appealing definition, on the other hand, they model well the LZ compression standard: each LZ-compressed text can be converted into an equivalent SLP with $\mathcal{O}(\log(N/n))$ multiplicative increase in length and in $\mathcal{O}(n \log(N/n))$ time (where N is the size of the decompressed text) [5, 40] while each SLP can be converted to an equivalent LZ-compressed text with a constant increase of length (and in linear time).

The approach of modelling compression by SLP in order to develop efficient algorithms turned out to be fruitful. Algorithmic problems for SLP-compressed strings were considered and successfully solved [25, 26, 36]. In particular, the recent state-of-the-art efficient algorithms for pattern matching in LZ compressed text essentially use the reformulation of LZ methods in terms of SLPs [11]. SLPs found their usage also in programme verification [14] as well as in verifying the bisimulation [6, 24]. Surprisingly, while SLPs were introduced mainly as a model for practical applications, they turned out to be useful also in strictly theoretical branches of computer science, for instance, in the word equations [37, 38]; in particular, the currently best PSPACE bound was obtained in this fashion by Plandowski [37].

For more information about the SLPs and their applications, please look at the recent survey of Lohrey [29].

1.2 Membership Problem

As SLPs are used both in theoretical and applied research in computer science, tools for dealing with them should be developed. In particular, one should be aware that whenever working with strings, these may be supplied as respective SLPs. Hence, all the usual string problems should be reinvestigated in the compressed setting, as the classical algorithms may not apply directly, be inefficient or the problems themselves may become computationally difficult.

From language theory point of view, the crucial questions stated in terms of strings, is the one of compressed string recognition. To be more precise, we consider classic membership problems, i.e. recognition by automata, generation by a grammar etc., in which the input is supplied as an SLP. We refer to such problems as *compressed membership problems*. These were first studied in the pioneering work of Plandowski and Rytter [39], who considered compressed membership problem for various formalism for defining languages. Already in this work it was observed that we should precisely specify, what part of the input is compressed. Clearly the input string, but what about the language representation (i.e. regular expression, automaton, grammar, etc.). Should it be also compressed or not? Both variant of the problem are usually considered, with the following naming convention: when only the input string is compressed, we use a name *compressed membership*, when also the language representation, we prepend *fully* to the name.

In years to come, the compressed membership problem was investigated for various language classes [15, 20, 21, 27, 28, 39]. Compressed word problem for groups and monoids [27, 31, 32], which can be seen as a generalisation of membership problem, was also considered.

Despite the large attention in the research community, the exact computational complexity of some problems remained open. The most notorious of those is the fully compressed membership problem (FCMP) for NFA, considered already in the work of Plandowski and Rytter [39]. Here, the compression of NFA is done by allowing it to have transitions by strings, instead of single letters, and representing these strings as SLPs.

It is relatively easy to observe that the compressed membership problem for the NFA is in P, however, the status of the fully compressed variant remained open for over a decade. Some partial results were already obtained by Plandowski and Rytter [39], who observed that it is in PSPACE and is NP-hard for the case of one-letter alphabet, both of these bounds being relatively natural. Moreover, they showed that this problem is in NP for some particular cases, for instance, for one-letter alphabet. Further work on the problem was done by Lohrey and Mathissen [30], who demonstrated that if the strings defined by SLP have polynomial periods, the problem is in NP, and when all strings are highly aperiodic, it is in P. Concerning the case of DFAs, it is known that even for a fixed regular language, the compressed membership is P-hard [3, 33], and no upper-bound better than PSPACE (which holds for NFAs as well) was known.

1.3 Our Results and Techniques

We establish the computational complexity of fully compressed membership problems for both NFAs and DFAs.

Theorem 1 *Fully compressed membership problem for NFA is in* NP, *for DFA it is in* P.

Our approach to the problem is essentially different than the ones of Plandowski and Rytter [39] and Lohrey and Mathissen [30]. The earlier work focused on the properties of strings described by SLPs and tried to use this knowledge in order to analyse the automaton and its input, hopefully this should result in an efficient algorithm. We take a completely different route: we analyse and change the way strings are described by the SLPs in the instance. That is, we focus on the SLPs, and not on the encoded strings. Roughly, our algorithm replaces a substring of two letters ab appearing in the input string with a new letter c throughout the instance and iterates this process. In this way strings in the instance are compressed 'in the same way'. The intuition behind this is that already testing the equality of SLPs is a nontrivial task [36] and consequently performing other operations on the SLPS is even more involved. When all strings are compressed in the same way, two appearances of the same string in the instance are represented in a canonical way, so, for instance, testing equivalence is trivial, and in fact other operations on the SLPs can be performed more efficiently.

In order to perform the compressions of a pair ab we first decompress the SLPs, so that appearances of ab can be easily identified. Since the decompressed text can be exponentially long, we do the decompression locally: we introduce explicit strings into the rules' bodies. Then, we compress these explicit strings uniformly. 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 text significantly: one 'round' of recompression, in which every pair of letters that was present at the beginning of the 'round' is compressed, should shorten the encoded strings by a constant factor.

1.4 Similar Techniques

While application of the idea of recompression to compressed membership problem is new, related approaches were previously employed, and somehow inspired the presented technique: most notably the idea of replacing short strings by a fresh letter and iterating this procedure was used by Mehlhorn et. al [34], in their work on data structure for equality testing for dynamic strings. They viewed this process as 'hashing' or 'signature building'. In particular their method can be straightforwardly applied to equality testing for SLPs, yielding a nearly cubic algorithm (as observed by Gawrychowski [9]); a faster implementation of the employed data structure was also proposed [1] and it leads to a nearly quadratic algorithm for the SLP equivalence testing [9]. However, the inside technical details of the construction makes the extension to FCMP problematic: while this method can be used to build 'canonical' SLPs for strings in the instance, there is no apparent way to control how these SLPs actually look like and how do they encode the strings. Thus it is unknown how to modify the NFA and its transitions in the process of building canonical SLPs.

The (mentioned earlier) recent approaches to FCMP by Mathissen and Lohrey [30] already implemented the idea of replacing strings with fresh letters as well as modifications of the instance such that such replacement is possible. Also, while it was

known that the problem is NP-hard already in the case of a one-letter alphabet [39], their algorithm used nondeterminsim only in this case; thus suggesting that the nondeterminism in the problem is strongly connected to long blocks of the same letter. However, the replacement was not iterated, and the newly introduced letters could not be further compressed. Also, they replaced blocks of letters chosen by some predefined criteria, in particular, only parts of the instance were compressed.

1.5 Other Applications of the Technique

The technique of local recompression was also successfully applied to fully compressed pattern matching [18], obtaining a faster (i.e. almost quadratic) algorithm for this problem. Furthermore, a variant of this method was also applied in the area of word equations. While not claiming any essentially new results, the recompression approach yielded much simpler proofs and algorithms with smaller memory consumption (though still polynomial) of many classical results in the area, like PSPACE algorithm for solving word equations, double exponential bound on the size of the solution, exponential bound on the exponent of periodicity, context-sensitiveness in case of $\mathcal{O}(1)$ variables, etc. [19].

2 Preliminaries

2.1 Straight Line Programmes

Formally, a *Straight line programme* (SLP) is a CFG such that each nonterminal has a unique rule and it cannot derive itself, i.e. the grammar is *acyclic*. Usually it is assumed that G is in a *Chomsky normal form*, i.e. each production is either of the form $X \to YZ$ or $X \to a$. By this assumption, strings defined by G's nonterminals have length at most 2^n, where n is the number of nonterminals of the grammar; since our algorithm will replace some substrings by shorter ones, none string defined by SLPs during the run of algorithm will exceed this length.

Remark 1 During the run of the CompMem, each string derived by a nonterminal has length at most 2^n.

We denote the unique string derived by nonterminal A by $\mathrm{val}(A)$ (like *value*). A *symbol* is either a letter or a nonterminal. The notion of val extends to strings of symbols in an obvious way.

2.2 Input

The instance of the fully compressed membership problem (FCMP) for NFA consists of an input string, represented by an SLP, and an NFA N, whose transitions are labelled by SLPs.

For our purposes it is more convenient to assume that all SLPs are given as a single context free grammar G with a set of nonterminals $\mathcal{X} = \{X_1, \ldots, X_n\}$, the

input string is defined by X_n and the NFAs transitions are labelled with nonterminals of G. While we require that the input grammar is in the Chomsky normal form, during the algorithm we allow the grammar to be in a slightly more general form, described by the following conditions:

$$\text{each } X_i \text{ has exactly one production, which has at most 2 noterminals,} \tag{1a}$$

$$\text{if } X_j \text{ appears in the rule for } X_i \text{ then } j < i, \tag{1b}$$

$$\text{if } \mathrm{val}(X_i) = \epsilon \text{ then } X_i \text{ does not appear in the rules' bodies.} \tag{1c}$$

The strings that appear in the rule's body appear *explicitly* in this rule, or alternatively they are called *explicit strings*; this notion is introduced to distinguish them from the substrings of $\mathrm{val}(X_i)$.

Without loss of generality we may assume that the input string starts and ends with designated, unique symbols, denoted as \$ and #. These are not essential, however, the first and last letter of $\mathrm{val}(X_n)$ need to be treated in a somewhat special manner, furthermore, also the transitions by \$ and # in the NFA are treated in a special way (for instance, there is a unique transition by each of them). Having special symbols for the first and last letter makes the analysis smoother.

2.3 Input Size, Complexity Classes

The size $|G|$ of the representation of grammar G is the sum of the lengths of G's rules' bodies (we count ϵ as occupying a single entry). The size $|N|$ of the representation of NFA N is the sum of number of its states and transitions. The size $|\Sigma|$ of alphabet Σ is simply the number of elements in Σ.

By npolytime (polytime) we denote the *non-deterministic polynomial running time* (deterministic, respectively), with respect to $|N|$, $|\Sigma|$, $|G|$ and n. As usual, depending on the context, this describes either the running time of a specific algorithm, or a class of algorithms running in such time; the context will always uniquely determine, which of this meaning applies. By NP (P, respectively) we denote the complexity classes of the decision problems solvable in npolytime (polytime, respectively).

The input instance size is polynomial in $|N|$, $|G|$, $|\Sigma|$ and n, which denotes the number of nonterminals in G. One of the crucial properties of our algorithm is that n only decreases during the run of the algorithm. For this reason we modify the input instance so that its size is polynomial in n alone:

Remark 2 Without loss of generality we may assume that initially $|N|$, Σ and $|G|$ are all at most n.

To satisfy this additional condition it is enough to add dummy nonterminals (with rules) that are not used anywhere in the instance. Clearly this increases the size of the instance by a constant factor.

2.4 Automata, Paths and Labels, Determinism

Since we investigate automata, proofs deal mainly with (accepting) paths for strings. The constructed NFAs have transitions labelled with either letters, or non-terminals

of G. That is $\delta \subseteq Q \times (\Sigma \cup \mathcal{X}) \times Q$. Consequently, a *path* $\mathcal{P}$ from state p_1 to p_{k+1} (with intermediate states $p_2, \ldots, p_k$) is a sequence $\alpha_1\alpha_2 \ldots \alpha_k$, where $\alpha_i \in \Sigma \cup \mathcal{X}$ and $\delta(p_i, \alpha_i, p_{i+1})$. We write that $\mathcal{P}$ *induces* such a list of labels. The val($\mathcal{P}$) *defined* by such a path $\mathcal{P}$ is simply val($\alpha_1 \ldots \alpha_k$). We also say that $\mathcal{P}$ is a path for a string val($\mathcal{P}$). A path is *accepting*, if it ends in an accepting state. We usually consider paths beginning in a starting state. A string w is accepted by N if there is an accepting path from the starting state for w.

Usually it is more convenient to represent path's list of labels as

$$u_{i_1} X_{i_2} u_{i_3} X_{i_4} \cdots X_{i_{m-1}} u_{i_m},$$

where each $u_{i_j} \in \Sigma^*$ is a string representing the consecutive letter labels and X_{i_j} represents a nonterminal label. Notice that u_{i_j} may be empty. The val($\mathcal{P}$) *defined* by such a path $\mathcal{P}$ is $\text{val}(\mathcal{P}) = u_{i_1} \text{val}(X_{i_2}) u_{i_3} \text{val}(X_{i_4}) \cdots \text{val}(X_{i_{m-1}}) u_{i_m}$.

We consider also DFAs with compressed labels. Let us comment, what 'determinism' means here: a NFA with compressed labels is *deterministic*, when for each state q and any two transitions from q labelled with α and α', the first letters of val(α) and val(α') are different. Other known (meaningful) definitions of determinism are polynomially equivalent to the given one.

2.5 Known Results

We use the following basic result: when the input string is over a one-letter alphabet the FCMP is in NP for NFA and in P for DFA.

Lemma 1 (cf. [39, Theorem 5]) *The FCMP restricted to the input string over an alphabet $\Sigma = \{a\}$ is in* NP *for NFA and in* P *for DFA.*

Proof Notice that if the input string w is over an alphabet $\{a\}$, no accepting path in the NFA for w may use transitions that denote strings having letters other than a. Thus, any such transitions can be deleted from N and we end up with an instance, in which $\Sigma = \{a\}$, i.e. also transitions in the NFA are labelled either with a letter a or by nonterminals defining some powers of a. Then the result of Plandowski and Rytter [39, Theorem 5] can be applied directly, claiming an NP upper-bound.

Their proof follows by an observation that when $\Sigma = \{a\}$ then an accepting path in the NFA exists if and only if the NFA satisfies an Eulerian-type condition: each state is entered and leaved the same number of times. Since each transition is used at most exponentially many times, a description of such a path can be guessed and then it can be verified whether it satisfies the condition and defines a word of appropriate length.

Consider now the deterministic automaton. As shown in the beginning, we can limit ourselves to transitions by powers of a. Since the automaton is deterministic, for each state there is at most one transition labelled with a power of a, and so the path for w cycles after at most n transitions. As the lengths of the cycle can be calculated in polytime, the whole problem can be easily checked in polytime. □

3 Basic Classifications and Outline of the Algorithm

In this section we present the outline of the algorithm for FCMP for NFAs. Its main part consist of *recompression*, i.e. replacing strings appearing in $\mathrm{val}(X_n)$ by shorter ones. In some cases, such replacing is harder, in other easier and identifying such hard cases is the first step in this section.

3.1 (Non)-Crossing Appearances, Maximal Blocks

We say that a string s has a *crossing appearance in a (unique string derived by) nonterminal* X_i with a production $X_i \to uX_jvX_kw$, if s appears in $\mathrm{val}(X_i)$, but this appearance is not contained in neither u, v, w, $\mathrm{val}(X_j)$ nor $\mathrm{val}(X_k)$. Intuitively, this appearance 'crosses' the symbols in $u\,\mathrm{val}(X_j)v\,\mathrm{val}(X_k)w$, i.e. at the same time part of s is in the explicit substring (u, v or w) and part is in the compressed strings ($\mathrm{val}(X_j)$ or $\mathrm{val}(X_k)$). This notion is similarly defined for nonterminals with productions of only one nonterminal, i.e. of the form $X_i \to uX_jv$, productions of the form $X_i \to u$ clearly do not have crossing appearances.

A string s has a *crossing appearance in the NFA* N, if there is a path in N inducing list of labels $\alpha_1\alpha_2$, where $\alpha_1, \alpha_2 \in \mathcal{X} \cup \Sigma$ with at least one of α_1, α_2 being a nonterminal, such that s appears in a $\mathrm{val}(\alpha_1\alpha_2)$, but this appearance is not contained in the $\mathrm{val}(\alpha_1)$, nor in $\mathrm{val}(\alpha_2)$. The intuition is similar as in the case of crossing appearance in a rule: it is possible that a string s is split between two transitions' labels. Still, there is nothing difficult in consecutive letter transitions, thus we treat such a case as a simple one.

We say that a pair of letters ab is a *crossing pair*, if ab has a crossing appearance of any kind. Otherwise, such a pair is *non-crossing*. Unless explicitly written, whenever we talk about crossing/non-crossing pair ab we assume that $a \neq b$.

We say that a letter $a \in \Sigma$ *has a crossing block*, if for some ℓ the a^ℓ has a crossing appearance; otherwise, a has no crossing block. This can be equivalently characterised by saying that a has a crossing block if and only if aa is a crossing pair.

The letters with crossing blocks and crossing pairs correspond to the intuitive notion of being 'hard' to compress.

The following lemma shows that while G may encode long strings, they have relatively few different short substrings and that they can be established efficiently (recall that n is the number of all noterminals in G, more precisely, they are $X_1, \ldots, X_n$).

Lemma 2 *There are at most* $2n$ *different letters with crossing blocks and at most* $|G| + 4n$ *different pairs of letters appearing in* $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$.

The set of letters with crossing blocks, the set of crossing pairs and the set of non-crossing pairs appearing in $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$ *can be computed in* polytime.

Proof Since a letter a has a crossing block if and only if aa is a crossing pair, it follows that if a has a crossing block then it is either the first, or the last letter of some $\mathrm{val}(X_i)$. Since there are at most n nonterminals, there are at most $2n$ letters with a crossing block. In order to calculate the set of letters with a crossing block, it

is enough to calculate the set of crossing pairs, as a has a crossing block if and only if aa is a crossing pair. Hence, in the rest of this proof all crossing pair can be of the form aa.

We estimate the total number of (different) pairs of letters appearing in $\mathrm{val}(X_1)$, $\ldots$, $\mathrm{val}(X_n)$. Consider first pairs that appear in some explicit string on the right-hand side of some production. Since the total length of explicit strings is $|G|$, there are at most $|G|$ such pairs. Other pairs are assigned to nonterminals $X_1, \ldots, X_n$: a pair ab is assigned to X_i, if it appears in $\mathrm{val}(X_i)$ and it does not appear in $\mathrm{val}(X_1)$, $\ldots$, $\mathrm{val}(X_{i-1})$. We show that at most four different pairs are assigned to each nonterminal. In this way, the total number of different pairs is at most $|G| + 4n$. Indeed, if ab is assigned to X_i with a production $X_i \to uX_jvX_kw$, then ab does not appear neither in u, v, w as in such case it was already accounted; nor in $\mathrm{val}(X_j)$, $\mathrm{val}(X_k)$, as in such case it is not assigned to X_i. Thus, there are four possibilities:

- a is the last letter of u and b is the first letter of $\mathrm{val}(X_j)$,
- a is the last letter of $\mathrm{val}(X_j)$ and b is the first letter of v,
- a is the last letter of v and b is the first letter of $\mathrm{val}(X_k)$,
- a is the last letter of $\mathrm{val}(X_k)$ and b is the first letter of w.

The cases, in which u or v is empty or there are less nonterminals on the right-hand side of the production, are similar.

The above description can be turned to a straightforward algorithm computing both the list of all non-crossing and crossing pairs appearing in $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$. First, the list of all pairs of letters with such appearances is calculated: clearly, it is enough to read every rule (for X_i) and store the pairs that appear in the explicit strings and the pairs that are assigned to X_i. Then, for each pair of letters it should be decided, whether it is crossing. To this end, we check, whether it has a crossing appearance in any nonterminal or in N, which can be done in polytime. Such pairs are crossing, other are non-crossing. Lastly, we filter out the pairs of the form aa from these lists, which gives the list of letters with crossing blocks. □

The notions of (non-) crossing pairs do not apply to aa, still, an analog can be defined: for a letter $a \in \Sigma$ we say that a^ℓ is a a's *maximal block of length* ℓ (or simply ℓ-*block*), if it appears in some string defined by some nonterminal and it is surrounded by letters other than a, formally, if there exist two letters $x, y \in \Sigma$, where $x \neq a \neq y$ and a nonterminal X_i, such that $xa^\ell y$ is a substring of $\mathrm{val}(X_i)$. Similarly to crossing pairs, it can be shown that there are not too many different maximal blocks of a.

Lemma 3 *For a letter a there are at most $|G| + 4n$ different lengths of a's maximal blocks in $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$. The set of these lengths can be calculated in* polytime.

The proof of Lemma 3 is similar to the proof of Lemm 2; however, Lemma 3 is not shown now, instead, we give a proof of a stronger Lemma 13 in a later section.

3.2 Outline of the Algorithm

Our algorithm is based on two main operations performed on strings encoded by G

Algorithm 1 Outline of the CompMem, which tests compressed membership

```
1:  while |val(X_n) > n| do
2:      perform the preprocessing   ▷ Reduces the number of crossing pairs to O(n)
3:      P ← list of non-crossing pairs in val(X_1), ..., val(X_n)
4:      P' ← list of crossing pairs in val(X_1), ..., val(X_n)
5:      for each ab ∈ P do
6:          compress ab, modify N accordingly
7:      for each ab ∈ P' do
8:          compress ab, modify N accordingly
9:      L ← letters in val(X_1), ..., val(X_n) without crossing blocks, except $, #
                                   ▷ Including letters introduced in line 6 and 8
10:     L' ← letters in val(X_1), ..., val(X_n) with crossing blocks, except $, #
                                   ▷ Including letters introduced in line 6 and 8
11:     for a ∈ L do
12:         compress blocks of a, modify N accordingly
13:     for a ∈ L' do
14:         compress blocks of a, modify N accordingly
15: Decompress X_n and solve the problem naively.
```

blocks compression of a For each a^ℓ that is an ℓ-block in $\mathrm{val}(X_n)$ and $\ell > 0$, replace all a's ℓ-blocks in $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$ by a fresh letter a_ℓ. Modify N accordingly.

pair compression of ab For two *different* letters a, b such that substring ab appears in $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$ replace each substring ab in $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$ by a fresh letter c. Modify N accordingly.

When the pair ab is crossing (or a has crossing blocks), the compression of ab (a blocks, respectively) is difficult; on the other hand, compression of non-crossing pairs (blocks of letters without crossing blocks, respectively) is easy. All this is explained in detail later in the section.

We denote the string obtained from w by a's blocks compression by $BC_a(w)$, and the string obtained by compression of a pair ab into c by $PC_{ab\to c}(w)$.

We adopt the following notational convention throughout rest of the paper: whenever we refer to a letter a_ℓ, it means that the last block compression was done for a and a_ℓ replaced a's ℓ-blocks.

The main idea behind the algorithm is that block compression and pair compression shorten the encoded texts significantly. The general schema is given in Algorithm 1.

The preprocessing, which is described in details later, modifies the instance slightly, so that the number of crossing pairs can be upper-bounded in terms of n: note that as crossing pairs can come from the NFA transitions, in general there can be as much as $\Omega(|N|)$ crossing pairs, which is more than the analysis can handle.

There are three important remarks to be made:

– there is no explicit non-deterministic operation in the code, however, it appears implicitly in the term 'modify the NFA accordingly' in lines 12 and 14. Roughly,

to perform such a modification, one needs to solve FCMP for string a^ℓ, and this is known to be NP-complete.
- the compression (both of pairs and blocks) is never applied to \$, nor to #. The markers were introduced so that we do not bother with strange behaviour when first or last letter is compressed, and so we do not touch the markers.
- CompMem, as presented, is deterministic, however some of its subprocedures are non-deterministic. This can make a false impression that it is in the class P^{NP}. However, we never alter the results returned by the non-deterministic procedures, and so CompMem is in fact in NP; this is formally stated and proved in the later sections.

Ideally, each letter of the input is compressed and so the $|\operatorname{val}(X_n)|$ halves in an iteration of the main loop. The worst case scenario is not far from the ideal behaviour.

Lemma 4 *There are $\mathcal{O}(n)$ executions of the loop in line 1 of* CompMem.

Proof Consider any 2 consecutive letters ab, where $a \neq \$$ and $b \neq \#$, appearing in the $\operatorname{val}(X_n)$ at the beginning of loop starting in line 1. We show that at least one of these two letters is compressed before the next execution of this loop. In this way, if we partition $\operatorname{val}(X_n)$ into blocks of 4 consecutive letters, each block is shortened by at least one letter in each iteration of the loop from line 1. Thus the length of $\operatorname{val}(X_n)$ decreases by a factor of $3/4$ in each iteration and so this loop is executed at most $\mathcal{O}(n)$ times, as in the input instance satisfies $|\operatorname{val}(X_n)| \leq 2^n$, see Remark 1.

Assume for the sake of contradiction that none of letters a, b is compressed during this iteration of the loop.

If $a \neq b$, then ab is going to be included in P or P' in line 3 or 4, respectively, depending on whether ab is crossing or not. Then CompMem will attempt to compress ab, either in line 6 or 8, and this fails only if one of the letters a or b was already compressed. This contradicts the assumption that none of the a, b was compressed.

So suppose now that $a = b$ and that none of these letters is compressed. In particular, none of these two appearances of a were compressed in line 6 or 8. Thus, a will be listed in L' in line 10 or in L in line 9, depending on whether it has crossing appearances or not. In the latter case, CompMem will compress the maximal block, in which these letters appear, in line 12, in the former in line 14. In either case, these letters a are compressed, a contradiction with the assumption that they were not. □

Remark 3 Notice that pair compression $PC_{ab\to b}$ is in fact introducing a new nonterminal with a production $c \to ab$, similarly BC_a introduces nonterminals a_ℓ with productions $a_\ell \to a^\ell$ (notice that in order to transform this production to Chomsky normal form, introduction of some other nonterminals is needed). Hence, CompMem creates new SLPs that encode strings from the instance. However, these new nonterminals are never expanded, they are always treated as individual symbols. Thus it is better to think of them as letters. Moreover, the analysis of running time of CompMem relies on the fact that no new nonterminals are introduced by CompMem.

4 Details

In this section we describe in detail how to implement the block compression and pair compression and how to modify the NFA. In particular, we are going to formulate the connections between NFA and SLPs preserved during CompMem.

4.1 Invariants

The invariants below describe the grammar kept by CompMem.

SLP 1 The set of used nonterminals is a subset of $\mathcal{X} = \{X_1, \ldots, X_n\}$ and the productions are of the form described in (1a)–(1c).

SLP 2 The nonterminal X_n has a production $X_n \to \$uX_{n-1}v\#$, where $u, v \in (\Sigma \setminus \{\$, \#\})^*$; \$, # are not used in other productions.

The following invariants represent the constraints on the NFA.

Aut 1 every transition of N is labelled by a single letter of Σ (*letter transition*) or by a nonterminal (*nonterminal transition*) that does not define ϵ, each nonterminal labels at most one transition. No transition is labelled with X_n.

Aut 2 there is a unique starting state that has a unique outgoing transition labelled by letter \$, and no incoming transitions; there is no other transition by \$. Similarly, there is a unique accepting state that has a unique incoming transition labelled by letter #, it does not have any outgoing transitions; there is no other transition by # in N.

CompMem will preserve (SLP 1)–(Aut 2), and we shall always assume that the input of the subroutines satisfies (SLP 1)–(Aut 2).

We assume that the input instance satisfies (SLP 1)–(Aut 2), moreover that the input grammar is in the Chomsky normal form. It is routine to transform (in polytime) the input instances not satisfying these conditions into equivalent instances that satisfy them; the only (seemingly) non-trivial one is the second requirement of (Aut 2) that there is a unique accepting state with a unique incoming transition for a DFA: to satisfy this condition we add two symbols $\#_1\#$ to the end of the X_n and create two new states in N, p_1 and p. Then we make a transition by $\#_1$ from each accepting state to p_1 and a unique transition from p_1 to p by #. Lastly, p becomes the unique accepting state.

4.2 Compression of Pairs

The compression of non-crossing pairs is intuitively easy: whenever these appear in strings encoded by G or on paths in N, they cannot be split between nonterminals or between transitions. So we replace their explicit appearances in the grammar and in the NFA. This is formalised and shown in the first subsection.

The compression of the crossing pairs does not directly follow this approach, however, for a fixed crossing pair ab we show that a simple transformation of the SLP makes ab a noncrossing pair, so that it consequently can be compressed using the known procedure. Thus, the compression of crossing pairs also can be done: first we fix a pair ab, then transform the instance, so that ab is noncrossing and then compress ab, using the already described procedure for compression of a noncrossing

Algorithm 2 PairComp(ab, c): compresses a non-crossing pair ab into c

```
for each production X_i → α do
    replace each explicit ab in α by c
for states p, q do
    if δ_N(p, ab, q) then
        put a transition δ_N(p, c, q)
```

pair. There can be as much as $\Omega(|N|)$ crossing pairs, however, a simple preprocessing reduces this number to $\mathcal{O}(n)$, see Lemma 10 later in this section. In particular, the transformation is used in total $\mathcal{O}(n)$ many times. This is described in detail in the second subsection.

4.2.1 Compression of Non-crossing Pairs

Consider a non-crossing pair ab. Since it is non-crossing, it can only appear in the explicit strings in the rules of G. Hence, compressing ab into a fresh letter c consists of replacing each explicit ab by c in rules' bodies. Still, ab can appear on a path in N. But since ab is non-crossing, this can be either wholly inside a nonterminal transition (and so compression was already taken care of), or on two consecutive letter transitions. This is also easy to handle: whenever there is a path from p to q by a string ab, we introduce a new letter transition by c from p to q. This description is formalised in PairComp (Algorithm 2).

To distinguish between the input and output G and N, we utilise the following convention: 'unprimed' names refer to the input (like G, X_i, N), while 'primed' symbols refer to the output (like G', X'_i, N'). This convention is used in lemmata concerning algorithms through the paper.

Lemma 5 PairComp(ab, c) *runs in* polytime *and preserves* (*SLP* 1)–(*Aut* 2). *When applied to a non-crossing pair of letters* ab, *where* $a, b \notin \{\$, \#\}$, *it implements the pair compression, i.e.* $\mathrm{val}(X'_i) = PC_{ab\to c}(\mathrm{val}(X_i))$, *for each* X_i.

N' *recognises* $\mathrm{val}(X'_n)$ *if and only if* N *recognises* $\mathrm{val}(X_n)$. *If* N *is a DFA, so is* N'.

If de *was a noncrossing pair in* G, N *and* $d \neq c \neq e$ *then* de *is also a noncrossing pair in* G', N'.

Proof The bound on the running time is obvious from the code.

Since PairComp only modifies the grammar by shortening some strings in the productions (it does not create ϵ-rules), and it does not affect \$ and # in the rules, (SLP 1)–(SLP 2) are preserved. The only modification in N is the introduction of new transition by a single letter (namely, by c) between states that are joined by a path for ab. Moreover, if there is a new transition $\delta_{N'}(p', c, q')$, then p has an outgoing production by $a \notin \{\$, \#\}$, and so it was not a starting or accepting state, and q had an incoming transition by $b \notin \{\$, \#\}$, and similarly it was not a starting nor accepting state. Thus (Aut 1)–(Aut 2) hold for N' as well. Notice that if N is deterministic, so is N': suppose that there are two different transitions starting with a letter d from state p in N'. If $d \neq c$, then these two transition are also present in N and they begin with

the same letter, which is not possible, as N is deterministic. If $d = c$, then either in N there are two transitions from p whose strings begin with a or there is a unique such transition, but $\delta(p, a)$ has two transitions whose strings begin with b. In both cases this is a contradiction.

We now show that N' recognises $\mathrm{val}(X'_n)$ if and only if N recognises $\mathrm{val}(X_n)$. To this end we demonstrate, how PairComp affects $\mathrm{val}(X_i)$:

Claim 1 *After performing* PairComp, *it holds that*

$$\mathrm{val}\big(X'_i\big) = PC_{ab\to c}\big(\mathrm{val}(X_i)\big). \tag{2}$$

Proof Notice that as $a \neq b$, $PC_{ab\to c}$ is well defined for each string.

The claim follows by a simple induction on the nonterminal's number: This is true when the production for X_i has no nonterminal on the right-hand side (recall the assumption that $a \neq b$), as in this case the pair compression on right hand side of the production for X_i is explicitly performed. When $X_i \to uX_jvX_kw$, then

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

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

$$\begin{aligned}
&PC_{ab\to c}\big(\mathrm{val}(X_i)\big)\\
&\quad = PC_{ab\to c}(u)PC_{ab\to c}\big(\mathrm{val}(X_j)\big)PC_{ab\to c}(v)PC_{ab\to c}\big(\mathrm{val}(X_k)\big)PC_{ab\to c}(u),
\end{aligned}$$

which shows that $PC_{ab\to c}(\mathrm{val}(X_i)) = \mathrm{val}(X'_i)$, ending the proof of the claim. □

The second claim similarly establishes, how the pair compression of a non-crossing pair affects the NFA. To be more precise, what happens to a string defined by a path in the NFA after applying pair compression to the underlying NFA.

Claim 2 *Consider a non-crossing pair ab and a path $\mathcal{P}$ in the NFA N, which defines a list of labels*:

$$u_{i_1}X_{i_2}u_{i_3}X_{i_4}\cdots X_{i_{m-1}}u_{i_m},$$

where each $u_{i_j} \in \Sigma^*$ is a string representing the consecutive letter labels and X_{i_j} represents a transition by a nonterminal transition. Then

$$\begin{aligned}
&PC_{ab\to c}\big(\mathrm{val}(\mathcal{P})\big)\\
&\quad = PC_{ab\to c}(u_{i_1})\,\mathrm{val}\big(X'_{i_2}\big)PC_{ab\to c}(u_{i_3})\,\mathrm{val}\big(X'_{i_4}\big)\cdots\mathrm{val}\big(X'_{i_{m-1}}\big)PC_{ab\to c}(u_{i_m}).
\end{aligned} \tag{3}$$

Proof Similarly as in Claim 1, notice that as ab is a non-crossing pair, the appearance of ab in the string defined by $\mathcal{P}$ cannot be split between a nonterminal and a string (or other nonterminal). Thus, replacement of pairs ab takes place either wholly inside string u or inside $\mathrm{val}(X_i)$. The former is done explicitly by $PC_{ab\to c}$, while (2) establishes the form of the latter. This ends claim's proof. □

After proving Claims 1–2, it is easy to show the main thesis of the lemma, i.e. that $\mathrm{val}(X'_n)$ is accepted by N' if and only if $\mathrm{val}(X_n)$ is accepted by N.

⟸ Suppose first that $\mathrm{val}(X_n)$ is accepted by N. Consider the accepting path $\mathcal{P}$ for $\mathrm{val}(X_n)$, represent it as a list of labels: $u_{i_1}X_{i_2}u_{i_3}X_{i_4}\cdots X_{i_{m-1}}u_{i_m}$, similarly as in Claim 2. Of course,

$$\mathrm{val}(\mathcal{P}) = \mathrm{val}(X_n) = u_{i_1}\,\mathrm{val}(X_{i_2})u_{i_3}\,\mathrm{val}(X_{i_4})\cdots\mathrm{val}(X_{i_{m-1}})u_{i_m}. \tag{4}$$

We will construct an accepting path $\mathcal{P}'$ in N' inducing a list of labels

$$PC_{ab\to c}(u_{i_1})X'_{i_2}PC_{ab\to c}(u_{i_3})X'_{i_4}\cdots X'_{i_{m-1}}PC_{ab\to c}(u_{i_m}). \tag{5}$$

Using (3) and recalling that $\mathrm{val}(\mathcal{P}) = \mathrm{val}(X_n)$ will be enough to conclude that $\mathrm{val}(\mathcal{P}') = PC_{ab\to c}(\mathrm{val}(X_n))$.

Notice that by the code of PairComp

- if there is a transition $\delta_N(p,d,q)$ for a letter $d \in \Sigma$ in N, then there is the same transition $\delta_{N'}(p,d,q)$ in N'.
- if there is a path from p to q for a string ab in N then there is a transition $\delta_{N'}(p,c,q)$ in N'.

Thus, by a trivial induction on the length of the string u, if $\delta_N(p,u,q)$ then also $\delta_{N'}(p, PC_{ab\to c}(u), q)$. The situation is similar for nonterminals: if there is a transition $\delta_N(p,X_i,q)$ in N, then there is an analogous transition $\delta_{N'}(p,X'_i,q)$ in N'. Thus, a path $\mathcal{P}'$ with the same starting and ending state as $\mathcal{P}$ and the list of labels as in (5) is inductively defined. Since the starting (accepting) state in N and N' coincide, this shows that $\mathrm{val}(\mathcal{P}')$ is accepted by N'.

⟹ Suppose now that a string $PC_{ab\to c}(\mathrm{val}(X'_n))$ is recognised by N'. Let the path of the accepting computation in N' be $\mathcal{P}'$, with a list of labels

$$u'_{i_1}X'_{i_2}u'_{i_3}X'_{i_4}\cdots X'_{i_{m-1}}u'_{i_m}.$$

Similarly to the previous case, we will inductively define an accepting path $\mathcal{P}$ in N with a list of labels

$$u_{i_1}X_{i_2}u_{i_3}X_{i_4}\cdots X_{i_{m-1}}u_{i_m}, \tag{6}$$

where u_{i_j} is obtained from u'_{i_j} by replacing each c by ab.

Notice that by PairComp,

- if there is a letter transition $\delta_{N'}(p,c,q)$ in N', there is a path from p to q for a string ab in N.
- if there is a letter transition $\delta_{N'}(p,d,q)$ for a letter $d \neq c$, there is the same transition $\delta_N(p,d,q)$ in N.

Thus, by a simple induction, we conclude that if there is path in N' from p to q for a string u'_{i_j}, then there is a path in N from p to q for a string u_{i_j}. Now observe that if there is a transition $\delta_{N'}(p, X'_i, q)$ in N', then there is an analogous transition $\delta_N(p, X_i, q)$ in N. This completes the construction of $\mathcal{P}$. Since the starting and accepting states in N and N' coincide, the constructed path $\mathcal{P}$ is also accepting, furthermore, $\mathcal{P}$'s list of labels is as in (6).

It is left to show that $\text{val}(\mathcal{P}) = \text{val}(X_n)$. Since $PC_{ab\to c}$ is a one-to-one function on string that do not contain c, and both $\text{val}(\mathcal{P})$ and $\text{val}(X_n)$ do not contain c, it is enough to show that $PC_{ab\to c}(\text{val}(\mathcal{P})) = PC_{ab\to c}(\text{val}(X_n))$. Notice that the latter equals $\text{val}(X'_n)$, by (2).

Using (3) we can conclude that

$$\begin{aligned} PC_{ab\to c}\big(\text{val}(\mathcal{P})\big) &= PC_{ab\to c}(u_{i_1})\,\text{val}\big(X'_{i_2}\big)\,PC_{ab\to c}(u_{i_3})\,\text{val}\big(X'_{i_4}\big)\cdots\text{val}(X_{i'_{m-1}})u'_{i_m} \\ &= u'_{i_1}\,\text{val}\big(X'_{i_2}\big)u'_{i_3}\,\text{val}\big(X'_{i_4}\big)\cdots\text{val}\big(X'_{i_{m-1}}\big)u'_{i_m} \\ &= \text{val}\big(\mathcal{P}'\big) \\ &= \text{val}\big(X'_n\big). \end{aligned}$$

Finally, consider the last claim of the lemma: let a pair $de \in P$ be a noncrossing pair. Since the whole modification to G is the replacement of pairs ab by a letter c, if de (where $d \neq c \neq e$) had no crossing appearances in G, then it does not have them in G'. Similarly, observe that transition by d and e in N and N' are the same, and if $\text{val}(X'_i)$ begins with e (ends with d) then also $\text{val}(X'_i)$ begins with e (ends with d, respectively). Thus, if de has a crossing appearance in N' then it also has it in N. □

4.2.2 *Crossing Pair Compression*

In this section we first show how to transform a crossing pair to a noncrossing one, so that PairComp can be applied to it. Then, we show how to perform a preprocessing after which the number of different crossing pairs is at most linear.

There are two possible reasons, why ab is a crossing pair: it may be that it is a crossing pair for some nonterminal, or it has a crossing appearance in the NFA. In both cases, the problem has something to do with the fact that b is the first letter of $\text{val}(X_i)$ or a is the last letter of some $\text{val}(X_i)$. To 'fix' this, we 'pop' such b and a from respective nonterminal: consider b and suppose that $\text{val}(X_i) = bw$. Then we modify G so that $\text{val}(X'_i) = w$ and modify N to reflect it, if X_i labels a transition in N. Clearly, after performing such operation for each nonterminal (including the symmetric procedure for a), b can still be a first letter of $\text{val}(X_i)$ (or a can be a last letter of $\text{val}(X_i)$, respectively); however, we show that ab is no longer a crossing pair and that these procedures can be easily performed.

Popping letters is performed in a bottom-up fashion, starting from X_1: when considering a rule $X_i \to \alpha$, if the first letter in $\text{val}(X_i)$ is b, we remove it and replace X_i by bX_i in all rules' bodies. It is easy to modify the NFA N accordingly: when there is a transition $\delta_N(p, X_i, q)$, we change it into a chain of two transitions: $\delta_{N'}(p, b, p_1)$ and $\delta_{N'}(p_1, X'_i, q)$. Symmetric treatment is applied to a, when it is the last letter of $\text{val}(X_i)$. This description is formalised in LeftPop (Algorithm 3).

Algorithm 3 LeftPop(b), pops leading b from each nonterminal

```
1:  for i ← 1..n do                                          ▷ Left-popping b
2:      let the rule for X_i be X_i → α
3:      if the first symbol of α is b then
4:          remove first letter (b) from α
5:          replace each X_i in rules' bodies by bX_i,
6:          if α = ϵ then
7:              remove X_i from rules' bodies
8:          if there is a transition δ_N(p, X_i, q) in N then    ▷ NFA modification
9:              remove transition δ_N(p, X_i, q)
10:             if α ≠ ϵ then
11:                 create new state p_1 in N,
12:                 set transitions: δ_N(p, b, p_1), δ_N(p_1, X_i, q)
13:             else
14:                 set transition δ_N(p, b, q)
```

Lemma 6 LeftPop *runs in* polytime *and when applied to* $b \notin \{\$, \#\}$ *it preserves* (*SLP* 1)–(*Aut* 2). *If* $\mathrm{val}(X_i) = bu$ *for some* $u \in \Sigma^*$ *then* $\mathrm{val}(X'_i) = u$; *otherwise* $\mathrm{val}(X'_i) = \mathrm{val}(X_i)$.

N' *accepts* $\mathrm{val}(X'_n)$ *if and only if* N *accepts* $\mathrm{val}(X_n)$. *If* N *is deterministic*, *so is* N'.

Proof The loop is executed n times, and also each line of the code can be performed in polytime, and so in total LeftPop runs in polytime.

Concerning the preservation of invariants: since the only operation performed on G is replacing nonterminal X_i by bX'_i and then deleting the first letter of the nonterminal, and nonterminals generating ϵ are explicitly removed from the rules, the resulting grammar is in the form (1a)–(1c). Notice that the invariant (SLP 1) is clearly preserved by the listed operations. As $b \notin \{\#, \$\}$, the first and last symbol of $\mathrm{val}(X_n)$ are not modified, and so (SLP 2) holds as well.

Let us move to the NFA invariants: the only change applied to NFA is the replacement of the transitions $\delta_N(p, X_i, q)$ by a path $\delta_{N'}(p, b, p_1)$, $\delta_{N'}(p_1, X_i, q)$, or by $\delta_{N'}(p, b, q)$, where b is the first letter of $\mathrm{val}(X_i)$. Clearly this does not affect (Aut 1)–(Aut 2). For the same reason, if N is deterministic, so is N'.

We show by induction on i, the number of the processed nonterminal that $\mathrm{val}(X_j) = \mathrm{val}(X'_j)$ if $j > i$ or the first letter of $\mathrm{val}(X_j)$ is not b; otherwise $\mathrm{val}(X_j) = b\,\mathrm{val}(X'_j)$. For the induction basis consider $i = 1$:

if the rule for X_1 is $X_1 \to bu$ for some $u \in \Sigma^*$ then this b is removed from the rule and each appearance of X_1 in the rules' bodies is replaced with bX'_1. Hence, $\mathrm{val}(X_1) = b\,\mathrm{val}(X'_1)$ and for each other nonterminal $\mathrm{val}(X_j) = \mathrm{val}(X'_j)$.

if the rule for X_1 is $X_1 \to u$ for some u not beginning with b then nothing is changed and so $\mathrm{val}(X_j) = \mathrm{val}(X'_j)$.

So consider an inductive step, let LeftPop consider the nonterminal X_{i+1}. We distinguish three copies of nonterminals now: the original one (so X_{i+1}), the one obtained after the processing of X_i but before X_{i+1} (those are denoted with primes,

i.e. X'_{i+1}) and the ones that are obtained after considering X_{i+1} (which are denoted with double prime, i.e. X''_{i+1}). By the inductive assumption $\mathrm{val}(X_{i+1}) = \mathrm{val}(X'_{i+1})$. If $\mathrm{val}(X_{i+1})$ does not begin with b, then the rule for X'_{i+1} does not begin with b either and so LeftPop performs no action and we are done. So suppose that $\mathrm{val}(X_{i+1})$ begins with b. By the inductive assumption X'_{i+1} defines the same string, and so begins with b as well. We claim that the first letter in the rule for X'_{i+1} is b: as $\mathrm{val}(X'_{i+1})$ begins with b, the only other option is that it begins with some nonterminal X'_k for $k < i$. But then a contradiction is easily obtained: $\mathrm{val}(X'_k)$ begins with b and so it can be concluded that $\mathrm{val}(X_k)$ begins with b as well, as by inductive assumption $\mathrm{val}(X_k) = \mathrm{val}(X'_k)$ or $\mathrm{val}(X_k) = b\,\mathrm{val}(X'_k)$, and both these strings begin with b. But as $\mathrm{val}(X_k)$ begins with b, by the code of LeftPop, X_k was replaced with bX'_k and this b is still in the rule for X'_{i+1}.

So the first letter in the rule for X'_{i+1} is b, and LeftPop removes this b from the rule and replaces each X'_{i+1} in the rules by bX''_{i+1}. Thus $\mathrm{val}(X_{i+1}) = \mathrm{val}(X'_{i+1}) = b\,\mathrm{val}(X''_{i+1})$. And in the rules, each X'_{i+1} was replaced with bX''_{i+1}, which evaluate to the same string, so values of other nonterminals have not changed.

We now show the second claim that N' accepts exactly the same strings as N. The only change done in the NFA is the replacement of transitions of the form $\delta_N(p, X_i, q)$ by a path inducing list of labels with b and X'_i, where b is the first letter of $\mathrm{val}(X_i)$, or by a transition $\delta_N(p, b, q)$, when $\mathrm{val}(X_i) = b$. Let us consider the former case, the latter is similar. Notice that $\mathrm{val}(X_i) = b\,\mathrm{val}(X'_i)$ and so the new path denotes the same string, as the replaced transition. Furthermore, the newly introduced state in the middle of this path has only one ingoing and outgoing transition. Since $b \notin \{\#, \$\}$, the starting and accepting states were not modified, and so the both automata recognise the same strings. Since $\mathrm{val}(X_n) = \mathrm{val}(X'_n)$ (again by $b \notin \{\#, \$\}$), this shows the claim. □

A symmetric variant RightPop of LeftPop, which pops the ending a from each nonterminal is easily defined. It satisfies the symmetric analogue of Lemma 6:

Lemma 7 RightPop *runs in time* polytime *and preserves* (*SLP* 1)–(*Aut* 2). *If* $\mathrm{val}(X_i) = ua$ *then* $\mathrm{val}(X'_i) = u$; *otherwise* $\mathrm{val}(X'_i) = \mathrm{val}(X_i)$.

N' *accepts* $\mathrm{val}(X'_n)$ *if and only if* N *accepts* $\mathrm{val}(X_n)$. *If* N *is deterministic*, *so is* N'.

Running both of LeftPop(b) and RightPop(a) makes a pair ab noncrossing.

Lemma 8 *After running* LeftPop(b) *and* RightPop(a) *the pair* ab *is non-crossing*.

Proof There are two different cases, why the pair ab is crossing: it is crossing in a rule or crossing in the NFA, consider first the former. As previously, let primed nonterminals, like X'_i denote the nonterminals in the instance after application of LeftPop(b) and RightPop(a) and X_i before this application.

Let X'_i be a nonterminal such that ab has an appearance in $\mathrm{val}(X'_i)$ that is not contained in any string or nonterminal of the rule for X'_i. There are three cases

– aX'_j appears in the rule and the first letter of $\mathrm{val}(X'_j)$ is b,

Algorithm 4 CrPairComp(ab, c), which compresses a crossing pairs ab

1: run LeftPop(b)
2: run RightPop(a)
3: run PairComp(ab, c)

– $X'_j b$ appears in the rule and a is the last letter of val(X'_j),
– $X'_j X'_k$ appears in the rule, where a is the last letter of val(X'_j) and b is the first letter of val(X'_k).

We shall consider only the first case, the other are shown in a similar way. Observe that by Lemma 6, if val(X'_j) begins with b then val(X_j) begins with b as well, as either val$(X_j) =$ val(X'_j) or val$(X_j) = b$ val(X_j). Hence LeftPop replaced X_j in each rule by bX'_j, and consequently the letter to the left of X'_j in the rule for X'_i cannot be a, as $a \neq b$.

So suppose that ab has a crossing appearance in the NFA. So there are two consecutive transitions α and β, such that the last letter of val(α) is a and the first of the val(β) is b. Furthermore, at least one of α, β is a nonterminal. Without loss of generality assume that $\beta = X'_i$ and let the transition be from state p to state q. As in the previous case, from the fact that the first letter of val(X'_i) is b we conclude that LeftPop modified X_i. In particular, the unique transition going to p is labelled with b, thus $\beta = b$, which is a contradiction. □

Now, it is enough to apply the pair compression for non-crossing pairs to each pair of the form ab. For convenience, we write the whole procedure for pair compression for crossing pairs in Algorithm 4.

Lemma 9 CrPairComp *runs in* polytime *and preserves* (*SLP* 1)–(*Aut* 2). *N' accepts* val(X'_n) *if and only if N accepts* val(X_n). *If N is deterministic, so is N'.*

It implements pair compression for ab, in the sense that val$(X'_n) = PC_{ab\to c}$(val(X_n)).

Proof The running time follows by Lemmata 5, 6 and 7.

Note that by Lemma 8 after the application of LeftPop(b) and RightPop(a) the pair ab is noncrossing.

By Lemmata 6–7, LeftPop(b) and RightPop(a) preserve (SLP 1)–(Aut 2) and after their application N recognises val(X_n) if and only if N' recognises val(X'_n). As ab is noncrossing, Lemma 5 guarantees that after the application of PairComp(ab, c), N recognises val(X_n) if and only if N' recognises val(X'_n).

Lastly, by Lemmata 6–7 the value of val(X_n) does not change when LeftPop(b) and RightPop(a) are applied and as ab is a noncrossing pair when PairComp(ab, c) is applied, Lemma 5 guarantees that val$(X'_n) = PC_{ab\to c}$(val(X_n)). □

We apply CrPairComp to each of the crossing pairs separately, and each such application increases the size of N and G. Thus it would be good to bound the number of crossing pairs in terms of n rather than in terms of $|N|$ and n. In general, this is

Algorithm 5 PreProc: preprocessing—reduces the number of crossing pairs

```
1:  for i ← 1..n − 1 do
2:      let the rule for X_i be X_i → α and a be the first letter in α
3:      remove leading a from α
4:      replace each X_i in the rules' bodies by aX_i
5:      if α = ϵ then
6:          remove X_i from rules' bodies
7:      if there is a transition δ_N(p, X_i, q) in N then          ▷ NFA modification
8:          remove transition δ_N(p, X_i, q)
9:          if α ≠ ϵ then
10:             create new state p_1 in N,
11:             set transitions: δ_N(p, a, p_1), δ_N(p_1, X_i, q)
12:         else
13:             set transition δ_N(p, a, q)
14:     if α ≠ ϵ then
15:         perform the symmetric actions for the last letter (b) of α
```

not possible, however, a simple preprocessing reduces the number of crossing pairs to $\mathcal{O}(n)$. It is enough to 'pop' the first and last letter from each of the nonterminals.

Lemma 10 PreProc (*Algorithm 5*) *runs in time* polytime *and preserves* (*SLP* 1)–(*Aut* 2). *For* $i < n$ *if* $\mathrm{val}(X_i) = aub$ *then* $\mathrm{val}(X_i') = u$; *if* $\mathrm{val}(X_i) \in \Sigma$ *then* $\mathrm{val}(X_i') = \epsilon$; *lastly* $\mathrm{val}(X_n) = \mathrm{val}(X_n')$.

N' *accepts* $\mathrm{val}(X_n')$ *if and only if* N *accepts* $\mathrm{val}(X_n)$. *If* N *is deterministic, so is* N'.

Proof The proof is similar to the proof of Lemma 6 and thus it is omitted. The only (slight) difference is that now we need to show that for each nonterminal X_i, when it is considered, its rule begins and ends with a letter. Still, this is easy: suppose that the rule for X_i begins with X_j. But then, when X_j was considered, X_j was replaced in all rules with aX_j, for some letter a, see line 4 of PreProc. In particular, this was done in the rule for X_i and so the rule for X_i cannot start with a nonterminal. Symmetric analysis shows that the rule cannot end with a nonterminal. □

As promised, after PreProc the number of crossing pairs is $\mathcal{O}(n)$:

Lemma 11 *After* PreProc *there are at most* $2n$ *crossing pairs.*

Proof Consider a nonterminal X_i. There are four ways in which X_i can contribute a crossing pair:

1. in some rule there is a substring aX_i (or X_jX_i);
2. there is a pair of consecutive transitions α and X_i in the NFA;
3. in some rule there is a substring X_ia (or X_iX_j);
4. there is a pair of consecutive transitions X_i and α in the NFA.

We claim that for a fixed nonterminal X_i, all cases of the form (1)–(2) yield at most one crossing pair, and all cases (3)–(4) also yield at most one crossing pair. Thus, this will yield $2n$ crossing pairs in total.

So consider a substring aX_i and $a'X_i$ that both appear in rules. Then both a and a' were obtained in the same way: they were introduced in the respective rules in line 4 of PreProc and there was no way to change them afterwards. Hence, $a = a'$ is the unique letter popped in the line 4 of PreProc. Note that this analysis shows also that a substring X_jX_i cannot appear in any rule. In the same way, consider the transition by X_i in the NFA N, let it be from p to q. But the line 11 of PreProc guarantees that there is a unique incoming transition to p, which is the same letter as the one popped in line 4 of PreProc, i.e. a. Consequently, cases (1)–(2) can introduce one crossing pair per nonterminal.

A symmetric analysis applies to (3)–(4). □

4.3 Blocks Compression

The block compression is very similar in spirit to pair compression, the only additional difficulty is the fact that blocks can be long, i.e. up to exponential. In case of letters without a crossing block, the compression can be done as in the case of noncrossing pairs, i.e. by replacing explicit blocks in G and adding some transitions to N. The case of a letter with a crossing block is reduced to the simpler case of letter without such a block: similarly to the compression of crossing pairs, we need to 'pop' letters from the beginning and the end of a nonterminal. However, this time popping one letter is not enough, we need to remove the whole a-prefix and a-suffix.

4.3.1 Compression of Noncrossing Blocks

Consider a that has no crossing block. Then every maximal block of a in $\mathrm{val}(X_1)$, $\ldots$, $\mathrm{val}(X_n)$ is an explicit substring in one of the rule' bodies; so we simply replace explicit a^ℓ by a fresh letter a_ℓ in rules' bodies, for each ℓ. Furthermore, as a's block cannot have a crossing appearance in N, when a^ℓ is a substring of a string defined by a path in N, then a^ℓ appears wholly inside a nonterminal transition, or a^ℓ labels a path using letter transitions only. The former case is taken care of by compression of a maximal blocks in G, and in the latter case for each a^ℓ and each pair of states p and q we check whether there is a path for a^ℓ from p to q using letter transitions only, which is done using the method from Lemma 1.

Note that BlockCompNcr(a) (Algorithm 6) uses nondeterministic subprocedures, which can raise doubts whether it is not a P^{NP} algorithm. However, the following lemma shows that it is still an NP algorithm.

Lemma 12 *Suppose that* BlockCompNcr *is applied for a letter* $a \notin \{\$, \#\}$ *without crossing blocks. Then it preserves* (*SLP* 1)–(*Aut* 2) *and properly implements maximal block compression, i.e.* $\mathrm{val}(X'_i) = BC_a(\mathrm{val}(X_i))$ *for each* X_i.

The operations in line 6 of BlockCompNcr *can be performed in* npolytime, *other operations can be performed in* polytime.

N *recognises* $\mathrm{val}(X_n)$ *if and only if* N' *recognises* $\mathrm{val}(X'_n)$ *for some nondeterministic choices. If* N *is DFA, so is* N'.

Algorithm 6 BlockCompNcr(a), which compresses a blocks when a has no crossing blocks

```
establish the lengths ℓ_1, ..., ℓ_k of a's maximal blocks in val(X_1), ..., val(X_n)
for each a^{ℓ_m} do
    for each production X_i → α do
        replace every explicit maximal block a^{ℓ_m} in α by a_{ℓ_m}
    for states p, q in N do
        if δ_N(p, a^{ℓ_m}, q) then        ▷ Check non-deterministically, see Lemma 1
            put a transition δ_N(p, a_{ℓ_m}, q)
```

If b that is not of the form a_ℓ for any ℓ had no crossing blocks in G, N, then it does not have them in G', N'.

This lemma is shown in a stronger version in the next section, as Lemma 17.

4.3.2 Removing Crossing Blocks of a Letter

It was already mentioned that a has a crossing block if and only if aa is a crossing pair. We know a method transforming a crossing pair to a noncrossing pair, see Lemma 5, however, it essentially assumes that the pair consists of two different letter: in short, when aX_i appears in the rule and we left-pop a letter a from X_i, then the pair aa is still crossing, whenever $\mathrm{val}(X_i)$ still begins with a. This is fixed in the most straightforward manner: we keep left-popping the letters from X_i, until the first letter of $\mathrm{val}(X_i)$ is different from a. In other words, it is enough to remove each nonterminal's a-prefix (and a-suffix). To be more precise: fix i and let $\mathrm{val}(X_i) = a^{\ell_i} u a^{r_i}$, where u does not start nor end with a. Then our goal is to modify G so that $\mathrm{val}(X_i') = u$. (If $\mathrm{val}(X_i)$ is a power of a, we simply give $u = \epsilon$ and $r_i = 0$.) This can be done in a bottom-up fashion, by two subprocedures, one of them removes the prefix, the other the suffix; these subprocedures work similarly as LeftPop and RightPop. We need to modify the NFA accordingly: it is enough to replace the transition labelled with X_i by path consisting of three transitions, labelled with a^{ℓ_i}, X_i' and a^{r_i}.

The removed a-prefixes and a-suffixes can be exponentially long, and so we store them in the rules in a succinct way, i.e. a^ℓ is represented as (a, ℓ); the size of representation of ℓ is $\mathcal{O}(\log \ell)$, i.e. $\mathcal{O}(n)$, see Remark 1. We say that such a grammar is in an *a-succinct* form. The NFA N might have transitions labelled with a^ℓ, which are stored in succinct way as well. We say that N satisfies *a-relaxed* (Aut 1), if its transitions are labelled by nonterminals, a single letter or by a^ℓ, where $\ell \leq 2^n$. The semantics of the new automaton should be clear: it can traverse the transition labelled with a^ℓ when consuming the a^ℓ from the beginning of the input word. Similarly, this automaton is deterministic, if for any two transitions, labelled with α and β, originating from the same state, the first letters of $\mathrm{val}(\alpha)$ and $\mathrm{val}(\beta)$ are different.

Before stating the appropriate algorithms, we show that even with such a succinct representation, the number of different lengths of maximal blocks of a is also linearly bounded, similarly as in Lemma 3.

Lemma 13 (stronger variant of Lemma 3) *For a letter a and a grammar G, which can be given in an a-succinct form, there are at most $|G| + 4n$ different lengths of a's maximal blocks in* $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$. *The set of these lengths can be calculated in* polytime.

Proof Consider first the maximal blocks of a that are fully contained within some rule's body. Then each symbol (a letter a or a string of letters a^ℓ, represented by one symbol) can be uniquely assigned to the maximal block to which it belongs; in particular, there are at most $|G|$ such blocks. To calculate the lengths of such blocks, it is enough to read the explicit strings in the rules, adding the appropriate lengths, which can be done in polytime, as these lengths are at most 2^n.

The other maximal blocks are the ones with the crossing appearances. Assign a maximal block to the nonterminal X_i with the smallest i, such that a^ℓ is a substring of X_i and a^ℓ has a crossing appearance in X_i. Then there are at most 4 such blocks assigned to this rule: suppose it is of the form $X_i \to uX_jvX_kw$, then a^ℓ stretches over u and $\mathrm{val}(X_j)$ or over $\mathrm{val}(X_j)$ and v or v and $\mathrm{val}(X_k)$ or $\mathrm{val}(X_k)$ and w. Hence, there are at most $4n$ such lengths of maximal blocks. To calculate the lengths of these crossing blocks it is enough to calculate first the length of the a-prefix and a-suffix of each nonterminal, which is done in a straightforward bottom-up manner. Then it is enough to look at the rules and calculate the lengths of the a blocks stretching over both explicit letters and nonterminals in the rule. This is easily doable in polytime. □

Lemma 14 *The* CutPref(a) (*Algorithm 7*) *for $a \notin \{\$, \#\}$ runs in* polytime *time and preserves* (*SLP* 1)–(*Aut* 2), *except that it a-relaxes* (*Aut* 1). *G' is in the a-succinct form.*

Let $\mathrm{val}(X_i) = a^{\ell_i}u_i$, *where u_i does not begin with a. After* CutPref(a) $\mathrm{val}(X'_i) = u_i$.

N accepts $\mathrm{val}(X_n)$ *if and only if N' accepts* $\mathrm{val}(X'_n)$. *If N is a DFA, so is N'.*

Algorithm 7 CutPref(a), pops the a-prefix from each nonterminal

```
1:  for i ← 1..n do                                          ▷ Cutting a-prefixes
2:      let the rule for X_i be X_i → α
3:      if the first symbol of α is a then
4:          remove the explicit a prefix, say a^{ℓ_i}, from α
5:          replace each X_i in rules' bodies by a^{ℓ_i} X_i
6:          if α = ϵ then
7:              remove X_i from rules' bodies
8:          if there is a transition δ_N(p, X_i, q) in N then   ▷ NFA modification
9:              remove transition δ_N(p, X_i, q)
10:             if α ≠ ϵ then
11:                 create new state p_1 in N,
12:                 set transitions: δ_N(p, a^{ℓ_i}, p_1), δ_N(p_1, X_i, q)
13:             else
14:                 set transition δ_N(p, a^{ℓ_i}, q)
```

Proof We first explain, how to calculate the a-prefix a^{ℓ_i} of $\mathrm{val}(X_i)$: since G is in a-succinct form, this might be non-obvious. It is enough to scan the explicit strings stored in the productions' right-hand sides, summing the lengths of the consecutive a's appearances. This clearly works in polytime also for G stored in an a-succinct form, as the powers of a may have length at most 2^n, see Remark 1, and so the length of their representation is linear in n (the correctness of this approach is shown later).

The running time of CutPref(a) is in polytime, as the loop has n iterations and all lines can be performed in polytime.

Concerning the preservation of the invariants: as in each rule there are at most two nonterminals and each nonterminal introduces at most one a's block to the rule (in line 5 of CutPref) in each rule of the grammar at most 2 maximal blocks of a are introduced. They may be long, however, in compressed form we treat them as singular symbols. In this way rules of G are stored in an a-succinct form, which was explicitly allowed. Then the a-prefix is removed and if X_i defines ϵ, it is removed from the right-hand sides of the productions. This does not affect the (SLP 1)–(SLP 2) (recall that a is not \$, neither #). Since the NFA is also changed, we inspect the invariants regarding N: introducing new states p_1 and replacing transition $\delta_N(p, X_i, q)$ by two transitions $\delta_{N'}(p, a^{\ell_i}, p_1)$, $\delta_{N'}(p_1, X, q_1)$ preserves the (Aut 1)–(Aut 2), with the exception that it a-relaxes (Aut 1); the same holds in the case, when $\alpha = \epsilon$ and the transition $\delta(p, X_i, q)$ is replaced with $\delta(p, a_i^{\ell}, q)$.

Notice that if N is deterministic, so is N': as already mentioned the only change done to N is the replacement of transition by X_i by a path of two transitions a^{ℓ_i} and X_i', such that the first letter of $\mathrm{val}(X_i)$ is a and the state in the middle have exactly one incoming and outgoing transition. This preserves determinism of the automaton: Let X_i label a transition from p to q and let p_1 be the new state in the middle. Then p_1 has one outgoing transition, furthermore, the first letter of the word defined by the label of the transition from p' to p_1 is a, so the same as it used to be for X_i which led from p to q. As N was deterministic, no other transition from p had a as the first letter, and so also no such transition, except the one to p_1, exists in N'.

To show the correctness, we prove by induction on i the two main claims of the lemma:

- CutPref correctly calculates the length of the a-prefix of $\mathrm{val}(X_i)$, i.e. ℓ_i,
- $\mathrm{val}(X_i) = a^{\ell_i}\,\mathrm{val}(X_i')$.

For $i = 1$ notice that the whole production for X_1 is stored explicitly, and so CutPref correctly calculates the a-prefix of $\mathrm{val}(X_1)$ and after its removal, $\mathrm{val}(X_1) = a^{\ell_1}\,\mathrm{val}(X_1')$. Furthermore, as each X_1 in the rules was replaced with $a^{\ell_1}\,\mathrm{val}(X_1')$, the $\mathrm{val}(X_j)$ for $j > i$ is not changed.

For the induction step, let $X_i \to uX_jvX_kw$. Then by the induction assumption:

$$\begin{aligned}\mathrm{val}(X_i) &= u\,\mathrm{val}(X_j)v\,\mathrm{val}(X_k)w\\ &= ua^{\ell_j}\,\mathrm{val}\big(X_j'\big)va^{\ell_k}\,\mathrm{val}\big(X_k'\big)w.\end{aligned}$$

There are cases to consider, depending on whether $\mathrm{val}(X_j') = \epsilon$ or not and whether $\mathrm{val}(X_k') = \epsilon$ or not. We describe the one with $\mathrm{val}(X_j') = \epsilon$ and $\mathrm{val}(X_k') \neq \epsilon$, other cases are treated in a similar way. Then the rule is rewritten as $ua^{\ell_j}va^{\ell_k}\,\mathrm{val}(X_k')w$.

Since by the inductive assumption, a is not the first letter of $\mathrm{val}(X'_k)$, the a prefix of $\mathrm{val}(X'_i)$ is the a-prefix of $ua^{\ell_j}va^{\ell_k}$. Thus it is correctly calculated by CutPref. As the last action, CutPref removes the a-prefix from the rule, which shows that $a^{\ell_i}\,\mathrm{val}(X'_i) = \mathrm{val}(X_i)$.

It is left to show that N accepts $\mathrm{val}(X_n)$ if and only if N' accepts $\mathrm{val}(X'_n)$. To this end, notice that the only modification to N is the replacement of the transition of the form $\delta_N(p, X_i, q)$ by a path labelled with a^{ℓ_i}, X'_i (or by a^{ℓ_i} alone). Furthermore, the vertex inside this path has only one incoming and one outgoing transition. The path labelled with a^{ℓ_i}, X'_i defines the string $a^{\ell_i}\,\mathrm{val}(X'_i)$, which was already shown to be $\mathrm{val}(X_i)$. It is left to observe that the newly introduced state in the middle of the path is not accepting, nor starting. Hence the starting (accepting) states of N and N' coincide, and so each string is accepted by N if and only if it is accepted by N'. □

A symmetric algorithm CutSuff, which removes the a-suffix, is also defined; it has similar properties as CutPref.

Lemma 15 *The* CutSuff(a) *for* $a \notin \{\$, \#\}$ *runs in* polytime *time and preserves* (*SLP* 1)–(*Aut* 2), *except that it* a*-relaxes* (*Aut* 1). G' *is in the* a*-succinct form.*

Let $\mathrm{val}(X_i) = u_i a^{r_i}$, *where* u_i *does not end with* a. *After* CutSuff(a) *the* $\mathrm{val}(X'_i) = u_i$.

N *accepts* $\mathrm{val}(X_n)$ *if and only if* N' *accepts* $\mathrm{val}(X'_n)$. *If* N *is a DFA, so is* N'.

Accordingly to the intuition provided at the beginning of this subsection, and similarly as in the case of the crossing pairs and procedures LeftPop, RightPop, it can be easily shown that after applying CutPref(a) and CutSuff(a) the letter a no longer has crossing blocks.

Lemma 16 *After application of* CutPref(a) *and* CutSuff(a), *for each nonterminal* X_i *neither first, nor last letter of* $\mathrm{val}(X_i)$ *is* a; *in particular, the letter* a *has no crossing blocks.*

Proof Observe that by Lemma 14 and 15 the letter a is not the first, nor the last letter of any $\mathrm{val}(X_i)$. Hence, it cannot has crossing blocks. □

Since after CutPref and CutSuff letter a no longer has crossing blocks, we may compress its maximal blocks using BlockCompNcr. Some small twitches are needed to accommodate the a-succinct form of G and the fact that N is a-relaxed: the nontrivial part of BlockCompNcr was the application of Lemma 1, which works for such large powers of a in npolytime, see Lemma 1. Other actions of BlockCompNcr generalise in a simple way.

Lemma 17 BlockCompNcr *can be extended, so that it applies to instances satisfying* (*SLP* 1)–(*SLP* 2) *with* G *in the* a*-succinct form and* a*-relaxed-*(*Aut* 1)–(*Aut* 2). *The output satisfies* (*SLP* 1)–(*Aut* 2) *and the claim of Lemma* 12 *applies to such an extension.*

Proof By Lemma 13, there are only $|G| + 4n$ possible lengths of a's maximal blocks and they can be calculated in polytime; hence line 1 of BlockCompNcr takes polytime even in this extended setting. All the loops in BlockCompNcr have only polynomially many iterations. All operations listed in BlockCompNcr are elementary and can be clearly performed in polytime, except replacing each a^ℓ by a_ℓ in a rule in line 4 and for the verification in line 6. For the former operation, it is enough to read the rules of the grammar: recall that a^ℓ is represented as a pair (a, ℓ). Since $\ell \leq 2^n$, addition of the lengths can be performed in polytime. The verification is more involved, we outline how to perform it: For given two states p, q and a string a^ℓ we want to verify, whether there is a path from p to q for a string a^ℓ. The transitions of the NFA are labelled either with single letters, powers of a (not larger then 2^n) or by nonterminals; however, from Lemma 16 it follows that for each nonterminal X_i nor the first, nor the last letter of $\mathrm{val}(X_i)$ is a. Thus, a path for a^ℓ can use only transitions labelled with powers of a (or a single letter a). Hence, our problem can be rephrased as a fully compressed membership problem for NFA over a unary alphabet: it is enough to

– restrict NFA N to transitions by powers of a,
– make p the unique starting state,
– make q the unique accepting state.

Since all considered powers a^r appear in strings defined by G, and so $r \leq 2^n$, see Remark 1. So each such a^r can be represented by an SLP of $\mathcal{O}(n)$ size. In particular, Lemma 1 is applicable here and so we can verify in npolytime whether there is a path from p to q for a word a^ℓ.

Concerning the preservation of invariants: we first show that the G' is not in the a-succinct form, nor N' is a-relaxed. When BlockCompNcr finishes its work, all maximal blocks of a are replaced, in particular, there are no succinct representations of a powers inside the grammar. Notice that there should be no transition labelled with a^ℓ in the NFA. To this end a new line at the end of BlockCompNcr should be added, so that all transitions by *powers* of a are removed.

Now we can return to showing the preservation of the invariants: since the only change to the productions consists of replacing maximal blocks of a by a single letter, (SLP 1)–(SLP 2) are preserved. Also, the only modifications to the NFA is the addition of new letter transitions. Thus, (Aut 1) holds. To see that also (Aut 2) holds, notice that if p receives new incoming (outgoing) transition in N', this transition is of the form a_ℓ and p had an incoming (outgoing, respectively) transition by a^ℓ in N. In particular, the starting and accepting state remain unaffected and no transition by \$ and # are introduced. Thus, also (Aut 2) holds for N'.

Notice that if N is deterministic, so is N': suppose that there are two different transitions in N' whose strings start with d. If d is not one of the new letters, then the same transitions were present in N, contradiction. So suppose it is one of the new letters, say a_ℓ. Observe that by Lemma 16 none of the strings $\mathrm{val}(X_i)$ started with a, and so after the block compression none of them starts with a_ℓ. So this means that there are two letter transitions starting from state p and labelled with a_ℓ. Thus, in N there are two paths for the string a^ℓ starting from p, which is a contradiction.

Before showing the main property of BlockCompNcr, we briefly comment on the last claim of the lemma: if BlockCompNcr is applied to a letter a and $b \neq a_\ell$ had no

crossing blocks before this application, then it also does not have after the application. This should be obvious: application of BlockCompNcr(a) introduces some new transitions to N, by letters other than b, changes transitions by a^ℓ into fresh letters (other than b) and changes a's blocks in G into fresh letters (again other than b); so all these operations do not influence, whether b has crossing blocks or not.

We proceed to the proof of the main property of BlockCompNcr: N accepts $\mathrm{val}(X_n)$ if and only if for some nondeterministic choices N' accepts $\mathrm{val}(X'_n)$. To this end we first show, how application of BlockCompNcr affects the words defined by G and the NFA N.

Claim 3 *After performing* BlockCompNcr, *it holds that*

$$\mathrm{val}\big(X'_i\big) = BC_a\big(\mathrm{val}(X_i)\big). \tag{7}$$

Claim 4 *Consider a letter a with no crossing blocks and a path $\mathcal{P}$ in N, which has a list of labels*:

$$u_{i_1} X_{i_2} u_{i_3} X_{i_4} \cdots X_{i_{m-1}} u_{i_m},$$

where each $u_{i_j} \in \Sigma^*$ is a string representing the consecutive letter labels and X_{i_j} represents a nonterminal transition, similarly as in Claim 2. Then

$$BC_a\big(\mathrm{val}(\mathcal{P})\big) = BC_a(u_{i_1})\,\mathrm{val}\big(X'_{i_2}\big)\,BC_a(u_{i_3})\,\mathrm{val}\big(X'_{i_4}\big)\cdots\mathrm{val}\big(X'_{i_{m-1}}\big)\,BC_a(u_{i_m}). \tag{8}$$

The proofs are analogous as the proofs of Claims 1–2 in Lemma 5 and are thus omitted. Notice that the properties stated in Claims 3–4 do *not* depend on the nondeterministic choices of BlockCompNcr.

It is left to show the main claims of the lemma: N recognises $\mathrm{val}(X_n)$ if and only if the NFA N' obtained for some non-deterministic choices recognises $\mathrm{val}(X'_n)$.

$\Leftarrow$ Suppose first that the N' accepts $\mathrm{val}(X'_n)$, using the path $\mathcal{P}'$. Clearly $\mathrm{val}(\mathcal{P}') = \mathrm{val}(X'_n)$. Let the list of labels on $\mathcal{P}'$ be

$$u'_{i_1} X'_{i_2} u'_{i_3} X'_{i_4} \cdots X'_{i_{m-1}} u'_{i_m}.$$

Let u_{i_j} be obtained from u'_{i_j} be replacing each a_{ℓ_k} with a^{ℓ_k}. We shall construct a path $\mathcal{P}$ in N, which has the same starting and ending as $\mathcal{P}'$ and induces a list of labels

$$u_{i_1} X_{i_2} u_{i_3} X_{i_4} \cdots X_{i_{m-1}} u_{i_m}.$$

Notice that

- if there is a transition $\delta_{N'}(p, X'_i, q)$ in N' then there is a transition $\delta_N(p, X_i, q)$ in N;
- if there is a transition $\delta_{N'}(p, b, q)$ for $b \neq a_{\ell_k}$ in N', then there is a transition $\delta_N(p, b, q)$ in N;
- if there is a transition $\delta_N(p, a_{\ell_k}, q)$ in N' for some a^{ℓ_k} that is a maximal block in one of $\mathrm{val}(X_1), \ldots, \mathrm{val}(X_n)$, then there is a path from p to q for a string a^{ℓ_k} in N.

Therefore, by an easy induction, $\mathcal{P}$ is a valid path in N, moreover, since $\mathcal{P}'$ is accepting, so is $\mathcal{P}$. It is left to demonstrate that $\mathcal{P}$ defines $\mathrm{val}(X_n)$: since BC_a is a one-to-one function on strings not containing letters of the form a_ℓ and both $\mathrm{val}(\mathcal{P})$ and $\mathrm{val}(X_n)$ do not contain such letters, it is enough to show that $BC_a(\mathrm{val}(\mathcal{P})) = BC_a(\mathrm{val}(X_n))$. By (7) it holds that $BC_a(\mathrm{val}(X_n)) = \mathrm{val}(X'_n)$. The value of $BC_a(\mathrm{val}(\mathcal{P}))$ is already known from (8), and so it is enough to show that

$$BC_a(u_{i_1})\,\mathrm{val}\big(X'_{i_2}\big)BC_a(u_{i_3})\,\mathrm{val}\big(X'_{i_4}\big)\cdots\mathrm{val}\big(X'_{i_{m-1}}\big)BC_a(u_{i_m}) = \mathrm{val}\big(X'_n\big).$$

But this is simply the fact that path $\mathcal{P}'$ defines $\mathrm{val}(X'_n)$, which holds by the assumption.

⟸ Suppose now that N accepts $\mathrm{val}(X_n)$. Consider the case in which BlockCompNcr always made a correct non-deterministic choice, i.e. that each time it correctly guessed in line 6.

Let the accepting path $\mathcal{P}$ in N has a list of labels

$$u_{i_1}X_{i_2}u_{i_3}X_{i_4}\cdots X_{i_{m-1}}u_{i_m},$$

where, similarly as in Claim 4, each u_{i_j} is a string representing the consecutive letter labels (u_{i_j} may be empty) and X_{i_j} represents a transition by a nonterminal transition. By the definition,

$$\mathrm{val}(X_n) = u_{i_1}\,\mathrm{val}(X_{i_2})u_{i_3}\,\mathrm{val}(X_{i_4})\cdots\mathrm{val}(X_{i_{m-1}})u_{i_m}.$$

Now consider the a's block compression applied to both side of this equality. By (7) and (8)

$$\mathrm{val}\big(X'_n\big) = BC_a(u_{i_1})\,\mathrm{val}\big(X'_{i_2}\big)BC_a(u_{i_3})\,\mathrm{val}\big(X'_{i_4}\big)\cdots\mathrm{val}\big(X'_{i_{m-1}}\big)BC_a(u_{i_m}).$$

We will construct an accepting path $\mathcal{P}'$ with a list of labels

$$BC_a(u_{i_1})X'_{i_2}BC_a(u_{i_3})X'_{i_4}\cdots X'_{i_{m-1}}BC_a(u_{i_m}).$$

Notice that $\mathrm{val}(\mathcal{P}') = \mathrm{val}(X'_n)$, and so construction of such path $\mathcal{P}'$ will conclude the proof. We iteratively transform $\mathcal{P}$ into $\mathcal{P}'$. Notice that

- if there is a transition $\delta_N(p, X_i, q)$ in N then there is a transition $\delta_{N'}(p, X'_i, q)$ in N';
- if there is a transition $\delta_N(p, b, q)$ for $b \neq a$ in N, then there is a transition $\delta_{N'}(p, b, q)$ in N';
- if there is a path in N from p to q for string a^ℓ that has maximal block in $\mathrm{val}(X_n)$, then there is a transition $\delta_{N'}(p, a_\ell, q)$ in N' (by the assumption that BlockCompNcr guessed correctly).

And by an easy induction $\mathcal{P}'$ is a valid path in N' and has the same starting and ending state as $\mathcal{P}$. Since $\mathcal{P}$ is accepting in N and the starting and accepting states in N and N' coincide, $\mathcal{P}'$ is an accepting path in N'. □

The CutPref, CutSuff and BlockCompNcr can be now used to implement the blocks' compression for an arbitrary letter, with crossing blocks or not.

Algorithm 8 BlockComp(a), which compresses a blocks

1: run CutPref(a)
2: run CutSuff(a)
3: run BlockCompNcr(a)

Lemma 18 *The* BlockComp(a) (*Algorithm 8*) *for* $a \notin \{\$, \#\}$ *preserves* (*SLP* 1)–(*Aut* 2) *and properly implements maximal block compression, i.e.* $\mathrm{val}(X'_n) = BC_a(\mathrm{val}(X_n))$.

It works in npolytime; *the only operation requiring nondeterminism is line 6 of* BlockCompNcr, *other operations can be performed in* polytime.

N *recognises* $\mathrm{val}(X_n)$ *if and only if* N' *recognises* $\mathrm{val}(X'_n)$ *for some nondeterministic choices. If* N *is DFA, so is* N'.

If b, *which is not of the form* a_ℓ *for any* ℓ, *had no crossing blocks in* G, N, *then it does not have them in* G', N'.

Proof This easily follows from Lemma 14, 15 and 17. □

4.4 Running Time and Correctness

Since the running time of each algorithm is npolytime, it is enough to show that the size of Σ, G and N are always polynomial in n (recall that n is unchanged throughout CompMem).

Lemma 19 *During* CompMem, *the sizes of* Σ, G, N *are polynomial in* n.

Proof We first bound the size of $|G|$. We show that at the beginning of each iteration of the main loop in CompMemeach right-hand side of the production has at most $64n + 16$ explicit letters, and that inside each iteration of the main loop of CompMem there are at most $16n + 4$ new symbols added (we exclude the letter replacing compressed strings). Notice that during the run of CompMem grammar G may be in succinct form, and accordingly we treat a^ℓ as one symbol.

These bounds hold when CompMem starts working, as the assumptions that G is in binary normal form implies that the right-hand sizes of the rules are of length at most 2. Let us fix a rule and consider, how many new letters may be introduced in this rule. By *introduced* we mean letters that were popped to this rule from some nonterminal, and not the letters that replaced pairs or blocks in this rule. There are five cases, in which new letters are introduced to a rule:

popping letters done by PreProc (line 4) In this way at most 4 new letters can be introduced to a rule.

popping letters by LeftPop (line 5) Each invocation of LeftPop introduces at most 2 new symbols to a rule. As LeftPop is used once for each crossing pair and there are at most $2n$ such pairs, see Lemma 11, in this way at most $4n$ new symbols were added per each iteration of the main loop of CompMem.

popping letters by RightPop Symmetric analysis, as in the previous case gives the same bound $4n$ on the number of letters introduced in this way to a rule.

cutting a prefix in CutPref (line 5) There are at most 2 new powers of a (all possibly in succinct form) that may be introduced to a rule in one invocation of CutPref. While these powers of a are written in a succinct representation, they will be all replaced by single letters in the later blocks compression for a. In total, CutPref is invoked for each letter with a crossing block, and there are at most $2n$ such letters, by Lemma 2. So there are at most $4n$ new symbols introduced in this way for one iteration of the main loop of CompMem.

cutting a suffix in CutSuff Analysis symmetric to the one for the cutting of suffix yields that at most $4n$ letters are introduced in one phase to a rule.

Hence, there are at most $16n + 4$ new letters introduced to the right-hand side of a production in each iteration of the main loop in CompMem. Still, the main task performed by CompMem is the compression: an argument similar as in the proof Lemma 4 can be used to show that the size of the explicit strings in the rules decreases by a factor of $3/4$ in each iteration of loop from line 1 in CompMem. Of course, the newly added letters may be unaffected by this compression. It is left to verify that $64n + 16$ is indeed the upper bound on the size of the right-hand size of a rule, let it be $X_i \to \alpha_i$:

$$\left|\alpha_i'\right| \leq \frac{3}{4} \cdot |\alpha_i| + (16n + 4) \leq \frac{3}{4} \cdot (64n + 16) + (16n + 4) = 64n + 16.$$

This proves the bound on $|G|$ at the end of each iteration of the main loop. Notice that as there at most $16n + 4$ new letters added to this rule, inside the phase the size of G is at most $80n^2 + 20n$.

We now turn our attention to the size of Σ. Again, consider the execution of CompMem and one iteration of the main loop. We show that there are polynomially many (in n) letters added in one such iteration. New letters are added to Σ when compression of pairs or block compression is applied. There are the following possibilities

compression of a non-crossing pair (line 2 of PairComp) Each compression of a non-crossing pair decreases the total length of explicit strings used in G by at least 1. Since the size of each right-hand side is at most $64n + 16$ at the beginning of the iteration and there are at most $16n + 4$ new letters added to a rule in each iteration, there can be at most $80n^2 + 20n$ such compressions, and so as many new letters added in this was.

compression of a crossing pair (call to PairComp made by CrPairComp) The crossing pairs compression is run for each of the crossing pairs, and there are at most $2n$ of them, see Lemma 11. So, at most $2n$ letters are introduced in this way.

block compression of a letter without a crossing block (line 4 of BlockCompNcr) The same argument as in the case of compression of noncrossing pairs applies.

block compression for of a letter with a crossing block (call to BlockCompNcr in BlockComp) There are at most $2n$ letters with crossing blocks, by Lemma 2, and each of them has at most $|G| + 4n$ different lengths of maximal blocks, by Lemma 13. Compressing all of them introduces at most $2n(|G| + 4n)$ new letters to Σ.

Since $|G|$ is polynomial in n, in each phase there are polynomially many (in n) letters introduced to Σ. Since the number of phases is $\mathcal{O}(n)$, see Lemma 4, the total size of $|\Sigma|$ is polynomial in n.

It is left to bound the size of $|N|$: as there are at most n non-terminal transitions, the size of transition function of N is at most $\mathcal{O}(|Q_N|^2 \cdot (|\Sigma| + n))$. So it is enough to bound the number of states of N by a polynomial in n. Note that without loss of generality we can assume that the input NFA has at most n states, see Remark 2. Consider when new states are added into the NFA, there are only five situations, in which this happens:

popping letters in PreProc (line 10) This introduces at most two states per nonterminal transition (one for popping the first letter and one for the last letter). As there are at most n nonterminal transitions, by (Aut 1), this adds at most $2n$ states.

left-popping letters in LeftPop (line 11) LeftPop introduces one state per nonterminal and there are at most n such transitions, by (Aut 1). Furthermore, LeftPop is invoked once per each crossing pair, i.e. at most $2n$ times, see Lemma 11. So, at most $2n^2$ states are introduced in this way.

right-popping letters in RightPop The same analysis as in the previous case yields that at most $2n^2$ states are introduced in this way.

cutting a prefix in CutPref (in line 11) CutPref introduces one state per nonterminal transition. By (Aut 1) there are at most n such transitions. So it is enough to estimate, how many times CutPref is invoked. CutPref is run for each letter with a crossing block, and there are at most $2n$ such letters, see Lemma 2. Thus, one iteration of the main loop of CompMem adds at most $2n^2$ states in total.

cutting a suffix in CutSuff The same analysis as in the previous case yields that at most $2n^2$ states are introduced in this way.

It is left to recall that by Lemma 4 the main loop of CompMem is run $\mathcal{O}(n)$ times, hence in total there are $\mathcal{O}(n^3)$ states added to N'. This ends the proof of the lemma. □

Using Lemmas 5–19 it is now possible to conclude that CompMem correctly solves the FCMP for NFA, in nondeterministic polynomial (in n) time. The only source of non-determinism is the one in Lemma 1, and so for DFA the corresponding problem can be solved deterministically.

Proof of Theorem 1 The proof follows by showing that CompMem properly verifies, whether $\mathrm{val}(X_n)$ is accepted by N and that CompMem runs in npolytime.

Let us first show correctness of CompMem. All subroutines of CompMem (non-deterministically) modify the instance, changing G, N and X_n into G', N' and X'_n (notice that the output depends on the non-deterministic choices). Let $N^{(i)}$, $G^{(i)}$, $X_n^{(i)}$ for $i = 1, \ldots, k$ be the consecutive obtained instances, with $i = 1$ representing the input instance. Then $N^{(i)}$ accepts $\mathrm{val}(X_n)^{(i)}$ if and only if for some non-deterministic choices the resulting $N^{(i+1)}$ accepts $\mathrm{val}(X_n)^{(i+1)}$. This is shown in Lemmata 5, 9, 10, 12, 18 (if some of the procedures are deterministic, then the output does not depend on any choices). So, if $N^{(1)}$ does not accept $\mathrm{val}(X_n^{(1)})$ also $N^{(k)}$ does not accept $\mathrm{val}(X_n^{(k)})$. On the other hand, if $N^{(1)}$ accepts $\mathrm{val}(X_n^{(1)})$, then there exits a sequence of instances (representing proper non-deterministic guesses), such that for each i the $N^{(i)}$ accepts $\mathrm{val}(X_n^{(i)})$. In particular, $N^{(k)}$ accepts $\mathrm{val}(X_n^{(k)})$ and as $|\mathrm{val}(X_n^{(k)})| < n$, $\mathrm{val}(X_n^{(k)})$ can be decompressed and acceptance by $N^{(k)}$ can be checked naively in polytime.

Now, we should show that the running time is in fact (non-deterministic) polynomial. Lemmata 5, 9, 10, 12, 18 claim that each of the subroutine runs in npolytime in the size of the current instance. However, by Lemma 19, the size of this instance is always polynomial in n. Furthermore, each of such application introduces a new letter to Σ, and we know by Lemma 19 that the final size of Σ is polynomial in n. Therefore these subroutines are run at most polynomially many (in n) times. Hence, the total running time is npolytime.

It is left to show that if the input is a DFA, CompMem can be determinised. Firstly, notice that by Lemmata 5, 9, 10, 12, 18, if the instance consisted of a DFA, each instance kept by CompMem is also a DFA.

The only non-deterministic choices in CompMem are performed when calling a subroutine for a fully compressed membership problem for a string over an alphabet consisting of a single letter (see Lemma 1). However, the same lemma states that when the input consists of a deterministic automaton, the problem is in P. Thus, there is no non-determinism in CompMem, when it is applied to a DFA. □

Acknowledgements I would like to thank Paweł Gawrychowski for introducing me to the topic, for pointing out the relevant literature [30, 34] and discussions [9] and to anonymous referees that pointed several shortcomings and whose comments helped in improving the presentation.

References

1. Alstrup, S., Brodal, G.S., Rauhe, T.: Pattern matching in dynamic texts. In: Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 819–828 (2000). doi:10.1145/338219.338645
2. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: pattern matching in Z-compressed files. In: SODA, pp. 705–714 (1994)
3. Beaudry, M., McKenzie, P., Péladeau, P., Thérien, D.: Finite moniods: from word to circuit evaluation. SIAM J. Comput. **26**(1), 138–152 (1997)
4. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: Randall, D. (ed.) SODA, pp. 373–389. SIAM, Philadelphia (2011)
5. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory **51**(7), 2554–2576 (2005)
6. Czerwiński, W., Lasota, S.: Fast equivalence-checking for normed context-free processes. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS, LIPIcs, vol. 8, pp. 260–271. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Wadern (2010)
7. Farach, M., Thorup, M.: String matching in Lempel-Ziv compressed strings. In: STOC, pp. 703–712. ACM Press, New York (1995)
8. Ferragina, P., Muthukrishnan, S., de Berg, M.: Multi-method dispatching: a geometric approach with applications to string matching problems. In: STOC, pp. 483–491 (1999)
9. Gawrychowski, P.: (2011). Personal communication
10. Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. In: Randall, D. (ed.) SODA, pp. 362–372. SIAM, Philadelphia (2011)
11. Gawrychowski, P.: Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA. LNCS, vol. 6942, pp. 421–432. Springer, Berlin (2011)
12. Gawrychowski, P.: Simple and efficient LZW-compressed multiple pattern matching. In: CPM. LNCS Springer, Berlin (2012)

13. Gawrychowski, P.: Tying up the loose ends in fully LZW-compressed pattern matching. In: Dürr, C., Wilke, T. (eds.) STACS, LIPIcs, vol. 14, pp. 624–635. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Wadern (2012)
14. Genest, B., Muscholl, A.: Pattern matching and membership for hierarchical message sequence charts. Theory Comput. Syst. **42**(4), 536–567 (2008). doi:10.1007/s00224-007-9054-1
15. Gąsieniec, L., Karpiński, M., Plandowski, W., Rytter, W.: Efficient algorithms for Lempel-Ziv encoding. In: Karlsson, R.G., Lingas, A. (eds.) SWAT. LNCS, vol. 1097, pp. 392–403. Springer, Berlin (1996)
16. Gąsieniec, L., Karpiński, M., Plandowski, W., Rytter, W.: Randomized efficient algorithms for compressed strings: the finger-print approach (extended abstract). In: Hirschberg, D.S., Myers, E.W. (eds.) CPM. LNCS, vol. 1075, pp. 39–49. Springer, Berlin (1996)
17. Gąsieniec, L., Rytter, W.: Almost optimal fully LZW-compressed pattern matching. In: Data Compression Conference, pp. 316–325 (1999)
18. Jeż, A.: Faster fully compressed pattern matching by recompression. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP. LNCS, vol. 7391, pp. 533–544. Springer, Berlin (2012)
19. Jeż, A.: Recompression: a simple and powerful technique for word equations. In: STACS 2013 Conference, LIPIcs. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik, Wadern (2013). Available at http://arxiv.org/abs/1203.3705
20. Jeż, A., Okhotin, A.: Complexity of equations over sets of natural numbers. Theory Comput. Syst. **48**(2), 319–342 (2011)
21. Jeż, A., Okhotin, A.: One-nonterminal conjunctive grammars over a unary alphabet. Theory Comput. Syst. **49**(2), 319–342 (2011)
22. Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., Arikawa, S.: Multiple pattern matching in LZW compressed text. In: Data Compression Conference, pp. 103–112 (1998)
23. Kosaraju, S.R.: Pattern matching in compressed texts. In: Thiagarajan, P.S. (ed.) FSTTCS. LNCS, vol. 1026, pp. 349–362. Springer, Berlin (1995)
24. Lasota, S., Rytter, W.: Faster algorithm for bisimulation equivalence of normed context-free processes. In: Královič, R., Urzyczyn, P. (eds.) MFCS. LNCS, vol. 4162, pp. 646–657. Springer, Berlin (2006). doi:10.1007/11821069_56
25. Lifshits, Y.: Solving classical string problems an compressed texts. In: Ahlswede, R., Apostolico, A., Levenshtein, V.I. (eds.) Combinatorial and Algorithmic Foundations of Pattern and Association Discovery, Dagstuhl Seminar Proceedings, vol. 06201. IBFI, Schloss Dagstuhl, Wadern (2006)
26. Lifshits, Y., Lohrey, M.: Querying and embedding compressed texts. In: Královič, R., Urzyczyn, P. (eds.) MFCS. LNCS, vol. 4162, pp. 681–692. Springer, Berlin (2006)
27. Lohrey, M.: Word problems and membership problems on compressed words. SIAM J. Comput. **35**(5), 1210–1240 (2006). doi:10.1137/S0097539704445950
28. Lohrey, M.: Compressed membership problems for regular expressions and hierarchical automata. Int. J. Found. Comput. Sci. **21**(5), 817–841 (2010)
29. Lohrey, M.: Algorithmics on SLP-compressed strings: a survey. Groups Complex. Cryptol. **4**(2), 241–299 (2012)
30. Lohrey, M., Mathissen, C.: Compressed membership in automata with compressed labels. In: Kulikov, A.S., Vereshchagin, N.K. (eds.) CSR. LNCS, vol. 6651, pp. 275–288. Springer, Berlin (2011). doi:10.1007/978-3-642-20712-9_21
31. Lohrey, M., Schleimer, S.: Efficient computation in groups via compression. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR. LNCS, vol. 4649, pp. 249–258. Springer, Berlin (2007). doi:10.1007/978-3-540-74510-5_26
32. MacDonald, J.: Compressed words and automorphisms in fully residually free groups. Int. J. Autom. Comput. **20**(3), 343–355 (2010)
33. Markey, N., Schnoebelen, P.: A PTIME-complete matching problem for SLP-compressed words. Inf. Process. Lett. **90**(1), 3–6 (2004)
34. Mehlhorn, K., Sundar, R., Uhrig, C.: Maintaining dynamic sequences under equality tests in polylogarithmic time. Algorithmica **17**(2), 183–198 (1997)
35. Navarro, G., Raffinot, M.: Practical and flexible pattern matching over Ziv-Lempel compressed text. J. Discrete Algorithms **2**(3), 347–371 (2004)
36. Plandowski, W.: Testing equivalence of morphisms on context-free languages. In: van Leeuwen, J. (ed.) ESA, LNCS, vol. 855, pp. 460–470. Springer, Berlin (1994). doi:10.1007/BFb0049431
37. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004). doi:10.1145/990308.990312

38. Plandowski, W., Rytter, W.: Application of Lempel-Ziv encodings to the solution of words equations. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP. LNCS, vol. 1443, pp. 731–742. Springer, Berlin (1998). doi:10.1007/BFb0055097
39. Plandowski, W., Rytter, W.: Complexity of language recognition problems for compressed words. In: Karhumäki, J., Maurer, H.A., Paun, G., Rozenberg, G. (eds.) Jewels Are Forever, pp. 262–272. Springer, Berlin (1999)
40. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. **302**(1–3), 211–222 (2003)