Compressed Membership for NFA (DFA) with Compressed Labels is in NP (P) *

Artur Jeż[1]

1 **University of Wrocław**
ul. Joliot-Curie 15, 50-383 Wrocław, Poland
aje@cs.uni.wroc.pl

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 introduced: the SLPs are recompressed, so that substrings of the input text 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 [21] and extending the partial result of Lohrey and Mathissen [14]; as this problem is known to be NP-hard, 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.

1998 ACM Subject Classification F.4.3 Formal Languages, F.4.2 Grammars and Other Rewriting Systems, F.2.2 Nonnumerical Algorithms and Problems, F.1.1 Models of Computation

Keywords and phrases Compressed membership problem, SLP, Finite Automata, Algorithms for compressed data

Digital Object Identifier 10.4230/LIPIcs.STACS.2012.136

1 Introduction

1.1 Compression and Straight-Line Programms

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 desperate, as it may seem: 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. Indeed, efficient algorithms for fundamental text operations (pattern matching,

* This work was partially supported by NCN grant number DEC-2011/01/D/ST6/07164, 2011–2014 and by a personal scholarship funded by the Human Capital Programme.

29th Symposium on Theoretical Aspects of Computer Science (STACS'12).
Editors: Christoph Dürr, Thomas Wilke; pp. 136–147
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SCIENCE

checking equality, etc.) are known for various practically used compression methods (LZ, LZW, etc.) [2, 3, 5].

Practical compression methods, like LZW or LZ variants, differ and so only algorithms for data compressed using them are also different. This leads to a plethora of algorithms for various compression variants and string operations. 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 instance can be simply seen as context-free grammars generating exactly one string.

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 the LZ compression standard: each LZ compressed text can be converted into an equivalent SLP with only logarithmic, with respect to decompressed data's size, overhead (and in polynomial time) while each SLP can be converted to an equivalent SLP with just a constant overhead (and in polynomial time).

The approach of modelling compression by SLP in order to develop efficient algorithms turned out to be fruitful. Algorithmic problems for SLP-compressed input strings were considered and successfully solved [10, 11, 18]. The recent state-of-the-art efficient algorithms for pattern matching in LZ and LZW compressed text essentially use the reformulation of LZW and LZ methods in terms of SLPs [2, 3]. SLPs found their usage also in programme verification [4, 9]. 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 [20, 19]; in particular, the currently best PSPACE bound was obtained in this fashion by Plandowski [19].

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 [21], 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 [5, 7, 12, 13, 21]. Compressed word problem for groups and monoids [12, 15, 16], which can be seen as a generalisation of membership problem, was also investigated.

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 [21]. 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 a long time. Some partial results were obtained by Plandowski and Rytter [21], who observed that it is in PSPACE and is NP-hard for the case of one-letter alphabet, both of these bounds being relatively simple. 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 [14], 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 [17, 1], and no upper-bound better than PSPACE 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 [21] and Lohrey and Mathissen [14]. The earlier work focused on the properties of strings described by SLPs. We take a completely different route: we analyse and change the way strings are described by the SLPs in instance. That is, we focus on the SLPs, and not on the encoded strings. Roughly, our algorithm aims at having all the strings in the instance compressed 'in the same way'. To achieve this goal, we decompress the SLPs. Since the compressed text can be exponentially long, we do this locally: we introduce explicit strings into the rules' bodies. Then, we recompress 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.

Other application of the technique

We stress that the idea of local decompression and recompression of SLP is new and promising: there is hope that it can be applied to other problem related to SLPs. In fact between the submission of this work and its acceptance the author successfully applied this technique in fully compressed pattern matching [6], obtaining a faster algorithm for this problem.

2 Preliminaries

2.1 Straight line programmes

Formally, a *Straight line programme* (SLP) is context free grammar G with a language consisting of exactly one string. 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; since our algorithm will replace some substrings by shorter ones, none string defined by SLPs during the run of algorithm will exceed this length.

We denote the string defined by nonterminal A by $\text{val}(A)$ (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 may be 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:

$$\text{each nonterminal has exactly one production, which is of the form} \tag{1a}$$

$$X_i \to uX_jvX_k \;\text{ or }\; X_i \to uX_jv \;\text{ or }\; X_i \to u, \quad \text{where } u, v \in \Sigma^* \text{ and } j, k < i, \tag{1b}$$

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

The strings u, v and their substrings appear *explicitly* in a rule, this notion is introduced to distinguish them from the substrings of $\text{val}(X_i)$.

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 $\text{val}(X_n)$ need to be treated in a somewhat special manner, furthermore, this applies to these letters' appearances in the NFA as well. 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 length of G's rules' bodies. 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.

The input (or, in general, current instance) size is polynomial in N, G, Σ 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.

By npolytime (polytime) we denote the class of algorithms running in non-deterministic (deterministic, respectively) polynomial time, and by NP (P, respectively) the corresponding complexity classes of the decision problems.

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} 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 $\text{val}(\mathcal{P})$ *defined* by such a path $\mathcal{P}$ is simply $\text{val}(\alpha_1 \ldots \alpha_k)$. We also say that $\mathcal{P}$ is a path for a string $\operatorname{val}(\mathcal{P})$. A path is *accepting*, if it ends in an accepting state. A string w is accepted by N if there is an accepting path from the starting state for w.

We consider also DFAs with compressed labels. Let us comment, what 'determinism' means here: a NFA with compressed labels is a *deterministic*, when for each state q and any two transitions from q labelled with α and α', the first letters of $\operatorname{val}(\alpha)$ and $\operatorname{val}(\alpha')$ are different. One could define determinism meaningfully in other ways, however, all these notions seem to be polynomially equivalent.

2.5 Known results

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

▶ **Lemma 2** (cf. [21, Theorem 5]). *The FCMP restricted to the input string over an alphabet $\Sigma = \{a\}$ is in* NP *for NFA and in* P *for DFA.*

The second claim is trivial and the first claim can be easily inferred from the result of Plandowski and Rytter [21, Theorem 5], who proved that this problem is in NP, when $\Sigma = \{a\}$, i.e. also transitions in the NFA are labelled by powers of a only. In such a case a path in NFA exists if and only if it satisfies an Eulerian-type condition: each state is entered and leaved the same number of times. Since each edge is used at most expontentially 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.

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 $\operatorname{val}(X_n)$ by shorter ones. In some cases, such replacing is harder, in other easier. It should be intuitively clear that it depends on the position of letters inside encoded strings: if a is a first or last letter of some $\operatorname{val}(X_i)$, then recompressing strings including a looks difficult, as such strings can be split into different nonterminals and recompression requires heavy modification of G, or even rebuilding of the NFA. On the other hand, if a letter a is only 'inside' strings encoded by nonterminals, its compression is done only 'inside' rules of G, which seems easy. Thus, before we state the algorithm, we firstly introduce classification of letters (and strings) into 'easy' and 'difficult' to compress.

3.1 Crossing appearances, types of letters, maximal blocks

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

A string w 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 w appears in a $\operatorname{val}(\alpha_1\alpha_2)$, but this appearance is not contained in the $\operatorname{val}(\alpha_1)$, nor in $\operatorname{val}(\alpha_2)$. The intuition is similar as in the case of crossing appearance in a rule: it is possible that a string w 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 different letters ab is a *crossing pair*, if ab has a crossing appearance of any kind. Otherwise, such a pair is *non-crossing*.

We say that a letter $a \in \Sigma$ is *left-outer* (*right-outer*), if there is a nonterminal X_i, such that a is the leftmost (rightmost, respectively) symbol in $\text{val}(X_i)$. A letter is *outer*, if it is left-outer or right-outer. Otherwise, the letter is *inner*. Notice that if a pair ab is crossing, then b is left-outer or a is right-outer. The outer letters 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 few outer letters.

▶ **Lemma 3.** *There are at most $2n$ different outer letters and at most $|G| + 3n$ different pairs of letters appearing in $\text{val}(X_1)$, …, $\text{val}(X_n)$.*

The set of outer letters, the set of crossing pairs and the set of non-crossing pairs appearing in $\text{val}(X_1)$, …, $\text{val}(X_n)$ can be computed in polytime.

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 $\text{val}(X_i)$. Similarly to crossing pairs, it can be shown that there are not too many different maximal blocks of a.

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

3.2 Outline of the algorithm

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

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

pair compression of ab For two *different* letters ab replace each of ab in $\text{val}(X_1)$, …, $\text{val}(X_n)$ by a fresh letter c. Modify N accordingly.

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.

There are two 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 4 and 10. Roughly, to perform such a modification, one needs to solve FCMP for string a^ℓ, and this is known to be NP-hard.
- 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.

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

Algorithm 1 Outline of the CompMem, which tests compressed membership

```
1: while $|\text{val}(X_n) > n|$ do
2:     while something changed do
3:         for $a$: inner letter do
4:             compress blocks of $a$, modify $N$ accordingly
5:         for non-crossing pair $ab$ in $\text{val}(X_n)$, $a, b \notin \{\$, \#\}$ do
6:             compress $ab$, modify $N$ accordingly
7:     $L \leftarrow$ list of outer letters, except $\$$ and $\#$
8:                                         ▷ Including letters introduced in line 4 and 6
9:     for $a \in L$ do
10:        compress blocks of $a$, modify $N$ accordingly
11:        for each $a_\ell b$ in $\text{val}(X_n)$ do
12:            compress $a_\ell b$, modify $N$ accordingly
13: Decompress $X_n$ and solve the problem naively.
```

▶ **Lemma 5.** *There are $\mathcal{O}(n)$ executions of the loop in line 1 of* CompMem.

▶ Remark. Notice that pair compression $PC_{ab \to b}$ is in fact introducing a new nonterminal with a production $c \to ab$, similarly BC_a. 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 $\mathcal{X} = \{X_1, \dots, X_n\}$ and the productions are of the form described in (1).

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.

4.2 Compression of non-crossing pairs and inner letters

The compression of non-crossing pairs and block compression for inner letters 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 this subsection

Consider a non-crossing pair ab. Since it is non-crossing, it can only appear in the 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.

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.

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

```
for each production $X_i \to \alpha$ do
    replace each explicit $ab$ in $\alpha$ by $c$
for states $p$, $q$ do
    if $\delta_N(p, ab, q)$ then put a transition $\delta_N(p, c, q)$
```

▶ **Lemma 6.** PairComp *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'.

We can apply the same approach to the inner letters block compression. However, in this case, the modification of N uses non-determinism.

Since a is an inner letter, it cannot appear as the last or first letter of any nonterminal, and so every maximal block of a in $\mathrm{val}(X_1)$, …, $\mathrm{val}(X_n)$ 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. Before considering the NFA, notice that as a is an inner letter, a^ℓ cannot have a crossing appearance in N, and no nonterminal defines a^ℓ. Hence, 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.

Algorithm 3 BlockComp(a), which compresses inner letter a blocks

```
establish the lengths $\ell_1, \dots, \ell_k$ of $a$'s maximal block
for each $a^{\ell_m}$ do
    for each production $X_i \to \alpha$ do
        replace every explicit maximal block $a^{\ell_m}$ in $\alpha$ by $a_{\ell_m}$
    for states $p$, $q$ in $N$ do
        if $\delta_N(p, a^{\ell_m}, q)$ then                ▷ Check non-deterministically, see Lemma 2
            put a transition $\delta_N(p, a_{\ell_m}, q)$
```

▶ **Lemma 7.** *Suppose that* BlockComp *is applied for inner letter* $a \notin \{\$, \#\}$. *Then it preserves (SLP 1)–(Aut 2) and properly implements maximal block compression, i.e.* $\text{val}(X_i') = BC_a(\text{val}(X_i))$ *for each* X_i.

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

Each of the new letters a_ℓ *is inner.* N *recognises* $\text{val}(X_n)$ *if and only if* N' *recognises* $\text{val}(X_n')$ *for some non-deterministic choices. If* N *is DFA, so is* N'.

4.3 Compression of outer letters and crossing pairs

Now, we turn our attention to the compression of outer letters and crossing pairs. The outline is as follows: we fix an outer letter a and modify the instance, so that a becomes inner. Then, BlockComp is applied to a. Next, we want to compress each pair of the form $a_\ell b$. Such a pair can be crossing, as b can be a left-outer letter. Thus, we modify the instance again, so that none of $a_\ell b$ is a crossing pair so afterwards we compress it using PairComp.

4.3.1 Transforming an outer letter to an inner letter

The reason, why a is an outer letter, is that it is the first or the last symbol in some $\text{val}(X_i)$. To make it an inner letter, it is enough to remove each nonterminal's a-prefix and a-suffix. To be more precise: fix i and let $\text{val}(X_i) = a^{\ell_i} u a^{r_i}$, where u does not start nor end with a. Then our goal is to modify G so that $\text{val}(X_i') = u$. (If $\text{val}(X_i)$ is a power of a, we simply give $u = \epsilon$ and $r_i = 0$.) This can be done in a bottom-up fashion, starting from X_1: it is enough to calculate and memorise the lengths of the a-prefixes and a-suffixes for consecutive nonterminals, see the operations in lines 1–4 of OutToIn. Then 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 $\mathcal{O}(\log \ell)$, that is, linear in n. We say that such a grammar is in an *a-succinct* form. The situation is similar for the NFA, as it 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 $\ell \leq 2^n$.

▶ **Lemma 8.** OutToIn(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 $\text{val}(X_i) = a^{\ell_i} u_i a^{r_i}$, *where* u_i *does not begin, nor end with* a. *After* OutToIn $\text{val}(X_i') = u_i$. *In particular the letter* a *is inner.*

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

Algorithm 4 OutToIn(a), which changes an outer letter a to an inner letter.

```
for $i = 1 \,.\,.\, n$ do
    let the production for $X_i$ be $X_i \to \alpha_i$
    let $a^{\ell_i}$, $a^{r_i}$ be the the explicit $a$-prefix and $a$-suffix of $\alpha_i$
    remove prefix $a^{\ell_i}$ and suffix $a^{r_i}$ from $\alpha_i$
    replace appearance of $X_i$ in rules' bodies by $a^{\ell_j} X_j a^{r_j}$
    if val$(X_i) = \epsilon$ then remove $X_i$ from rules' bodies
    if there is a nonterminal transition $\delta_N(p, X_i, q)$ in $N$ then
        remove transition $\delta_N(p, X_i, q)$
        if val$(X_i) \neq \epsilon$ then
            create new states $p_1$, $q_1$ in $N$,
            set transitions: $\delta_N(p, a^{\ell_i}, p_1)$, $\delta_N(p_1, X_i, q_1)$, $\delta_N(q_1, a^{r_i}, q)$
        else create transition $\delta_N(p, a^{\ell_i}, q)$
```

Since after OutToIn a is no longer an outer letter, we may compress its maximal blocks using BlockComp. Some small twitches are needed to accommodate the a-succinct form of G and the fact that N is a-relaxed: the non-trivial part of BlockComp was the application of Lemma 2, which works for such large powers of a in npolytime, see Lemma 2. Other actions of BlockComp generalise in a simple way.

▶ **Lemma 9.** BlockComp *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 7 applies to such an extension.*

4.3.2 Crossing pair compression

By Lemma 9 all letters a_ℓ are inner. However, a pair of the form $a_\ell b$ can still be crossing and this can happen only when b is a left-outer letter. We try to fix it by ensuring that such a b is not a left-outer letter. To do so, we 'pop' one letter from the beginning of each nonterminal (that is, all left outer letters): let $\text{val}(X_i) = bw$ for $b \in \Sigma$, we modify G so that $\text{val}(X_i') = w$. Clearly, after such operations there are some (perhaps other) left-outer letters in G. Still, we show that none $a_\ell b$ is crossing.

Popping letters is performed similarly to the removal of the a-prefix, i.e. in a bottom-up fashion, starting from X_1: when considering a rule $X_i \to \alpha$, we remove the first letter from α, say b, 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 $\delta_{N'}(p_1, X_i', q)$. The whole operation is not performed on X_n, as the letter \$ is not going to be compressed anyway. This description is formalised in Algorithm 5.

▶ **Lemma 10.** Pop *runs in time* polytime *and preserves (SLP 1)–(Aut 2). Let* $\text{val}(X_i) = bu$*, where* $b \in \Sigma$*, then* $\text{val}(X_i') = u$ *for* $i < n$ *and* $\text{val}(X_n') = \text{val}(X_n)$*. After running* Pop*, pairs of the form* $a_\ell b$ *appearing in* $\text{val}(X_n)$ *are non-crossing.*

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

Now, it is enough to apply the pair compression for non-crossing pairs to each pair of the form $a_\ell b$. For convenience, we write the whole procedure for pair compression for crossing pairs in Algorithm 6.

Algorithm 5 Pop, pops the first letter from each nonterminal

```
for $i \leftarrow 1 \,..\, n-1$ do                                   ▷ Popping letters
    let $X_i \to \alpha$ and $b$ be the first letter of $\alpha$
    remove first letter ($b$) from $\alpha$
    replace each $X_i$ in rules' bodies by $bX_i$,
    if $\alpha = \epsilon$ then remove $X_i$ from rules' bodies
    if there is a transition $\delta_N(p, X_i, q)$ in $N$ then           ▷ NFA modification
        remove transition $\delta_N(p, X_i, q)$
        if $\alpha \neq \epsilon$ then
            create new state $p_1$ in $N$, set transitions: $\delta_N(p, b, p_1)$, $\delta_N(p_1, X_i, q)$
        else set transition $\delta_N(p, b, q)$
```

Algorithm 6 CrPairComp(a), which compresses crossing pairs $a_\ell b$

```
run Pop on each letter
for each $a_\ell$ do
    for each $a_\ell b$ appearing in val($X_n$) do
        run PairComp($a_\ell b, c$)
```

▶ **Lemma 11.** CrPairComp *runs in* polytime *and preserves (SLP 1)–(Aut 2). It implements pair compression for* $a_\ell b$*, in the sense that* $\mathrm{val}(X'_i) = PC_{a_\ell b \to c}(\mathrm{val}(X_i))$ *for each* 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'*.*

4.4 Running time

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 Algorithm 1).

▶ **Lemma 12.** *During Algorithm 1, the sizes of* Σ, G, N *are polynomial in* n*.*

Using Lemmas 6–12 it is now possible to conclude that Algorithm 1 correctly solves the FCMP for NFA, in nondeterministic polynomial (in n) time. The only source of nondeterminism is the one in Lemma 2, and so for DFA the corresponding problem can be solved deterministically.

Acknowledgements

I would like to thank Paweł Gawrychowski for introducing me to the topic and for pointing out the relevant literature.

References

1 Martin Beaudry, Pierre McKenzie, Pierre Péladeau, and Denis Thérien. Finite moniods: From word to circuit evaluation. *SIAM J. Comput.*, 26(1):138–152, 1997.

2 Paweł Gawrychowski. Optimal pattern matching in LZW compressed strings. In Dana Randall, editor, *SODA*, pages 362–372. SIAM, 2011.

3 Paweł Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. In Camil Demetrescu and Magnús M. Halldórsson, editors, *ESA*, volume 6942 of *LNCS*, pages 421–432. Springer, 2011.

4 Blaise Genest and Anca Muscholl. Pattern matching and membership for hierarchical message sequence charts. *Theory of Computing Systems*, 42(4):536–567, 2008.

5 Leszek Gąsieniec, Marek Karpiński, Wojciech Plandowski, and Wojciech Rytter. Efficient algorithms for Lempel-Ziv encoding. In Rolf G. Karlsson and Andrzej Lingas, editors, *SWAT*, volume 1097 of *LNCS*, pages 392–403. Springer, 1996.

6 Artur Jeż. Faster fully compressed pattern matching by recompression. *CoRR*, 1111.3244, 2011.

7 Artur Jeż and Alexander Okhotin. One-nonterminal conjunctive grammars over a unary alphabet. *Theory of Computing Systems*, 49(2):319–342, 2011.

8 Rastislav Královič and Paweł Urzyczyn, editors. *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28–September 1, 2006, Proceedings*, volume 4162 of *LNCS*. Springer, 2006.

9 Sławomir Lasota and Wojciech Rytter. Faster algorithm for bisimulation equivalence of normed context-free processes. In Královič and Urzyczyn [8], pages 646–657.

10 Yury Lifshits. Solving classical string problems an compressed texts. In Rudolf Ahlswede, Alberto Apostolico, and Vladimir I. Levenshtein, editors, *Combinatorial and Algorithmic Foundations of Pattern and Association Discovery*, volume 06201 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2006.

11 Yury Lifshits and Markus Lohrey. Querying and embedding compressed texts. In Královič and Urzyczyn [8], pages 681–692.

12 Markus Lohrey. Word problems and membership problems on compressed words. *SIAM Journal of Computing*, 35(5):1210–1240, 2006.

13 Markus Lohrey. Compressed membership problems for regular expressions and hierarchical automata. *International Journal of Foundations of Computer Science*, 21(5):817–841, 2010.

14 Markus Lohrey and Christian Mathissen. Compressed membership in automata with compressed labels. In Alexander S. Kulikov and Nikolay K. Vereshchagin, editors, *CSR*, volume 6651 of *LNCS*, pages 275–288. Springer, 2011.

15 Markus Lohrey and Saul Schleimer. Efficient computation in groups via compression. In Volker Diekert, Mikhail V. Volkov, and Andrei Voronkov, editors, *CSR*, volume 4649 of *LNCS*, pages 249–258. Springer, 2007.

16 Jeremy MacDonald. Compressed words and automorphisms in fully residually free groups. *International Journal of Automation and Computing*, 20(3):343–355, 2010.

17 Nicolas Markey and Ph. Schnoebelen. A ptime-complete matching problem for slp-compressed words. *Inf. Process. Lett.*, 90(1):3–6, 2004.

18 Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In Jan van Leeuwen, editor, *ESA*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.

19 Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *Journal of the ACM*, 51(3):483–496, 2004.

20 Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998.

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