Prime Normal Form and Equivalence of Simple Grammars

Cédric Bastien[1], Jurek Czyzowicz[1], Wojciech Fraczak[1,2], and Wojciech Rytter[3,4] ⋆

[1] Dépt d'informatique, Université du Québec en Outaouais, Gatineau PQ, Canada
[2] IDT Canada Inc., Ottawa ON, Canada
[3] Inst. of Informatics, Warsaw University, Warsaw, Poland
[4] New Jersey Institute of Technology, USA

Abstract. A prefix-free language is a prime if it cannot be decomposed into a concatenation of two prefix-free languages. We show that we can check in polynomial time if a language generated by a simple context-free grammar is a prime. Our algorithm computes a canonical representation of a simple language, converting its arbitrary simple grammar into Prime Normal Form (PNF); a simple grammar is in PNF if all its nonterminals define primes. We also improve the complexity of testing the equivalence of simple grammars. The best previously known algorithm for this problem worked in $O(n^{13})$ time. We improve it to $O(n^7 \log^2 n)$ and $O(n^5 \operatorname{polylog} v)$ deterministic time, and $O(n^4 \operatorname{polylog} n)$ randomized time, where n is the total size of the grammars involved, and v is the length of a shortest string derivable from a nonterminal, maximized over all nonterminals. Our improvement is based on a version of Caucal's algorithm from [1].

1 Introduction

An important question in language theory is, given a class of languages, find a canonical representation of any language of this class. Such a representation often permits to solve various decidability problems related to a given class of languages, such as equivalence of languages, non-emptiness, etc. Most often the canonical representation of the language is given by a special form of its grammar, called a normal form. In this paper, we give an algorithm converting a simple grammar into its equivalent, unique representation in a form of so-called Prime Normal Form (PNF). The canonical form of simple grammar was studied by Courcelle, c.f. [2]. The crucial question that our algorithm is confronted with, is whether a simple language is prime, i.e., not decomposable into a concatenation of two non-trivial prefix-free languages.

⋆ The research of the first three authors was supported by NSERC and the research of the fourth author was supported by the grants KBN 4T11C04425 and ITR-CCR-0313219.

In general, the canonical representation of any type of language may be substantially larger than its original grammar. This is also the case for simple languages. Hence verifying the equivalence of simple languages by means of canonical representations may be inefficient. The equivalence problem for simple context-free grammars is a classical question in formal language theory. It is a nontrivial problem, since the inclusion problem for simple languages is undecidable. A. Korenjak and J. Hopcroft, see [12, 8], proved that the equivalence problem is decidable and they gave the first, doubly exponential time algorithm solving it. Their result was improved by D. Caucal to $O(n^3v(G))$ time, see [1]. The parameter n is the size of the simple grammar and $v(G)$ is the length of a shortest string derived from a nonterminal, maximized over all nonterminals. Caucal's algorithm is exponential since $v(G)$ can be exponential with respect to n. Y. Hirshfeld, M. Jerrum, and F. Moller gave the first polynomial $O(n^{13})$ time algorithm for this problem in [9]. We call it the HJM algorithm.

In the second part of the paper we design an algorithm based on a version of Caucal's algorithm, that has a better complexity than HJM. More precisely, our algorithm works in time $O(n^7 \log^2 n)$. On the other hand a variation of our algorithm works in time $O(n^5 \operatorname{polylog}(v(G)))$, thus beating the complexity of Caucal's algorithm, e.g., for $v(G) \in \Omega(n^3)$. Similarly as the HJM algorithm, we apply the techniques used in the algorithmic theory of compressed strings, based on Lempel-Ziv string encoding. The idea of such an encoding is that, instead of representing a string explicitly, we design a context-free grammar generating the string as a one-word language. As the combinatorial complexity of such a grammar can be significantly smaller than the length of the word, it may be considered as a succinct representation of the word. Such encodings were recently considered by researchers, mainly in the context of efficient pattern matching. There is one problem in this field which is of particular interest to us — the *compressed first mismatch problem* (*First-MP*). Given two strings encoded by a grammar, *First-MP* looks for the position of the first symbol at which the strings differ. Polynomial time algorithms for computing *First-MP* were given independently in [9] and [15], in very disjoint settings. More powerful algorithms were given in [11], where a more complicated problem of fully compressed string-matching was solved. For the purpose of this paper, we will use the result from [13], which we adopted to obtain a faster algorithm.

Simple languages are applied by IDT Canada to perform packet classification at wire speed. Classes of packets are described with the aid of simple languages, and their recognition is made by a so-called Concatenation State Machine, an efficient version of a stateless pushdown automaton. As shown in [5], there is a one-to-one correspondence between Concatenation State Machines and simple grammars. In order to store large sets of classification policies in memory, it is necessary to reuse their common parts. A natural way to do this consists in decomposing simple languages into primes, each of which is stored in memory only once. When a new classification policy is added to memory, we verify if its prime factors are already stored in the data base. The algorithms described in this paper are used to decompose classification policies into primes and to identify primes for reuse.

2 Simple languages

A context-free grammar $G = (\Sigma, N, P)$ is composed of a finite set Σ of *terminals*, a finite set N of *nonterminals* disjoint from Σ, and a finite set $P \subset N \times (N \cup \Sigma)^*$ of *production rules*. For every $\beta, \gamma \in (N \cup \Sigma)^*$, if $(A, \alpha) \in P$, then $\beta A \gamma \to \beta \alpha \gamma$. A *derivation* $\beta \xrightarrow{*} \gamma$ is a finite sequence $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ such that $\beta = \alpha_0$, $\gamma = \alpha_n$, and $\alpha_{i-1} \to \alpha_i$ for $i \in [1, n]$.

For every sequence of nonterminals $\alpha \in N^*$ of a grammar $G = (\Sigma, N, P)$, we denote by $L_G(\alpha)$ the set of terminal strings derivable from α, i.e., $L_G(\alpha) \stackrel{\mathtt{def}}{=} \{w \in \Sigma^* \mid \alpha \xrightarrow{*} w\}$. Often, if G is known from the context, we will write $L(\alpha)$ instead of $L_G(\alpha)$.

A grammar $G = (\Sigma, N, P)$ is in *Greibach normal form* if for every production rule $(A \to \alpha) \in P$, we have $\alpha \in \Sigma N^*$. A grammar $G = (\Sigma, N, P)$ is a *simple context-free grammar* (simple grammar) if G is a Greibach normal form grammar and such that whenever $A \to a\,\alpha_1$ and $A \to a\,\alpha_2$, for a same $a \in \Sigma$, then $\alpha_1 = \alpha_2$.

A language $L \subseteq \Sigma^*$ is a *simple language* (also called *s-language*) if $L = \{\varepsilon\}$ (where ε denotes the empty word) or if there exists a simple grammar $G = (\Sigma, N, P)$ such that $L_G(A) = L$, for some $A \in N$. The definition implies that every nonterminal of a simple grammar defines a simple language. Since simple languages are prefix codes and are closed by concatenation, the family of simple languages under concatenation constitutes a free monoid with $\{\varepsilon\}$ as unit. Thus, every non-trivial simple language L (i.e. $L \neq \{\varepsilon\}$ and $L \neq \emptyset$) admits a unique decomposition into *prime* (i.e. undecomposable, non-trivial) simple languages, $L = P_1 P_2 \ldots P_n$.

3 Prime Normal Form for simple grammars

In this section we give an algorithm converting any simple grammar to its canonical representation called *Prime Normal Form*. A simple grammar is in Prime Normal Form (PNF) if each of its nonterminals represents a prime. We will use the following algebraic notation for left and right division in the free monoid of prefix codes. If $L = L_1 L_2$ for some prefix codes L, L_1, L_2, then by $L_1^{-1}L$ we denote L_2 and by LL_2^{-1} we denote L_1. We call L_1 a left divider and L_2 a right divider of L.

Let L be a prefix code and $L = P_1 P_2 \ldots P_n$ be its decomposition into primes. Prime P_n will be called *final prime* of L, and it will be denoted by $f(L)$. In particular, if L is a prime, then $f(L) = L$.

Lemma 1. *Let $G = (\Sigma, N, P)$ be a simple grammar. For every $X \in N$, there exists $Y \in N$, such that $f(L(X)) = L(Y)$.*

Proof. Let $w \in L(X)f(L(X))^{-1}$, and $X \xrightarrow{*} w\alpha$ be the leftmost derivation in G, with $\alpha \in N^+$. Since $L(\alpha) = f(L(X))$ and $L(\alpha)$ is a prime, α consists of a single nonterminal, i.e., $\alpha \in N$. $\square$

Let $w_0\,\alpha_0 \to \ldots \to w_i\,\alpha_i \to \ldots \to w_n\alpha_n$ be the leftmost derivation $X \xrightarrow{*} w$, with $w_0 = \varepsilon$, $\alpha_0 = X$, $w_n = w$, $\alpha_n = \varepsilon$, $w_i \in \Sigma^*$, and $\alpha_i \in N^*$, for $i \in [0, n]$. We are interested in the subsequence $\pi(X, w) = Y_0, Y_1, \ldots, Y_j$ of $\alpha_0, \alpha_1, \ldots, \alpha_n$, which consists of those elements of $\alpha_0, \ldots, \alpha_n$ that are single nonterminals. E.g., for the leftmost derivation of $abcdef \in L(X)$:

$$\underline{X} \to a\,YY \to ab\,\underline{Y} \to abc\,\underline{Y} \to abcd\,YZ \to abcde\,\underline{Z} \to abcdef$$

we have $\pi(X, abcdef) = X, Y, Y, Z$.

Definition 1. *Let $G = (\Sigma, N, P)$ be a simple grammar. We define relation $\mathcal{D}$ over $N \cup \{\varepsilon\}$ as follows. $(X, Y) \in \mathcal{D}$ if and only if:*

- *there exists a rule $(X \to a\alpha Y)$ in P for some $a \in \Sigma$ and $\alpha \in N^*$, or*
- *$Y = \varepsilon$ and there exists a rule $(X \to a)$ in P for some $a \in \Sigma$.*

Relation $\mathcal{D}$ can be seen as a digraph $(N \cup \{\varepsilon\}, \mathcal{D}, \varepsilon)$ with sink ε. In a digraph with a sink, vertex v is called a *d-articulation point* of vertex u if and only if v is present on every path from u to the sink. It was shown in [4] that the order of first (or last) occurrences of the d-articulation points of a vertex v is the same in all paths from v to the sink. Thus, it is natural to represent the set of all d-articulation points for a given vertex v as an ordered list of vertices, $(u_0, u_1, \ldots, u_n)$, where $u_0 = v$ and u_n is the sink.

In [4], it was shown that a prefix code L is prime if and only if the initial state v_1 of the minimal deterministic automaton for L does not have any d-articulation point except sink and v_1 itself. Moreover, the list of d-articulation points $(v_1, v_2, \ldots, v_n)$ corresponds to the prime decomposition of L, the factors being the languages defined by automata having v_i as the initial state and v_{i+1} as the final state (with all outgoing transitions of the final state removed), for $i \in [1, n)$, respectively.

Lemma 2. *For every path π from X to ε in $\mathcal{D}$ there exists a word $w \in L(X)$, such that $\pi = \pi(X, w)$. Conversely, for every $w \in L(X)$, $\pi(X, w)$ defines a path from X to ε in $\mathcal{D}$.*

We say that a grammar $G = (\Sigma, N, P)$ is *reduced* if there is no two different nonterminals defining the same language, i.e., for all $X, Y \in N$, if $L(X) = L(Y)$ then $X = Y$. By Lemma 1, the set of nonterminals $F(X) \stackrel{\text{def}}{=} \{Y \in N \mid L(Y) = f(L(X))\}$ is nonempty. If the underlying grammar is reduced then $F(X)$ consists of a single nonterminal which, by convenient abuse of notation, will be denoted by $f(X)$.

Theorem 1. *Let $G = (\Sigma, N, P)$ be a reduced simple grammar. For every $X \in N$, $L(X)$ is prime if and only if X does not have d-articulation points in $\mathcal{D}$ except sink and X itself. Moreover, if $Y \in N$ is a d-articulation point of X then $L(Y)$ is a right divider of $L(X)$.*

Proof. By Lemma 1, since G is reduced, every derivation starting in X is of form $X \xrightarrow{*} w' f(X) \xrightarrow{*} w$. Thus, for every $w \in L(X)$, $\pi(X, w)$ contains $f(X)$, i.e., $f(X)$ is a d-articulation point of X in $\mathcal{D}$.

Let Y be a d-articulation point of X in $\mathcal{D}$. By Lemma 2, every derivation starting in X passes by Y, thus Y is a d-articulation point for X in the (infinite) deterministic automaton for X, which implies that $L(Y)$ is a right divider of $L(X)$, cf. [6]. □

Theorem 2. *Given a reduced simple grammar $G = (\Sigma, N, P)$, we can find $f(X)$ for all $X \in N$ in linear time.*

Proof. By Theorem 1, the non-terminal $f(X)$ is exactly the second last d-articulation point for X in $\mathcal{D}$. Calculating $f(X)$ for all $X \in N$ can be done in linear time, by using an algorithm for finding dominators in flow graphs, cf. [7]. □

The algorithm for transforming a simple grammar $G = (\Sigma, N, P)$ into PNF, called *PNF*(G, S), is presented in Figure 1.

Input: Simple grammar $G = (\Sigma, N, P)$ and $S \in N^+$.

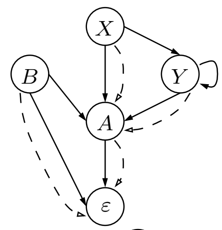
Output: Simple grammar G' in PNF and $S' \in N^+$, such that $L_G(S) = L_{G'}(S')$.

1. Reduce G.
   Find redundant nonterminals by checking if $L(X) = L(Y)$, for all $X, Y \in N$.
   Each redundant nonterminal is substituted in P and in S, and removed from N.
2. For every $X \in N$, find $f(X) \in N$.
   Construct the digraph $\mathcal{D}$ and find the second-last d-articulation point for X.
   If for every $X \in N$, $X = f(X)$, then **return** (G, S).
3. Construct a new grammar $G' = (\Sigma, N, P')$ and new S':
   Define morphism $h : N \mapsto N^*$ as: $h(X) \stackrel{\text{def}}{=} \begin{cases} X & \text{if } X = f(X) \\ Xf(X) & \text{otherwise.} \end{cases}$
   Set S' to $h(S)$, and P' as follows, for $a \in \Sigma$, $X, Y \in N$, $\alpha \in N^*$:
   (a) If $(X \to a\alpha) \in P$ and $X = f(X)$, then $(X \to ah(\alpha))$ is in P'.
   (b) If $(X \to a\alpha f(X)) \in P$ and $X \neq f(X)$, then $(X \to ah(\alpha))$ is in P'.
   (c) If $(X \to a\alpha Y) \in P$, $X \neq f(X)$, and $Y \neq f(X)$, then $(X \to ah(\alpha)Y)$ is in P'.
4. Set G to G', S to S' and go to 1.

Fig. 1. Algorithm *PNF*(G, S)

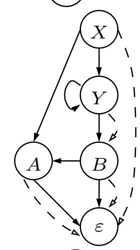
We present an example of the execution of the algorithm. The input consists of a simple grammar $G = \{(X \to aAA), (X \to bYY), (Y \to aY), (Y \to bBA), (A \to a), (B \to aXA), (B \to b)\}$, and a simple language represented as a word $S = XA$ over nonterminals of G. We obtain the grammar in PNF while keeping track of the decomposition of S. For each iteration, we give the value of S, the grammar G, the digraph $\mathcal{D}$ (solid lines), the d-articulation tree (dotted lines), and the values $f(x)$ for $x \in \{X, Y, A, B\}$ and $h(x)$ for $x \in \{X, Y, A, B, S\}$.

$$\textbf{Iteration 1:}\quad \begin{aligned} S &= XA \\ X &= aAA + bYY \\ Y &= aY + bBA \\ A &= a \\ B &= aXA + b \end{aligned}$$



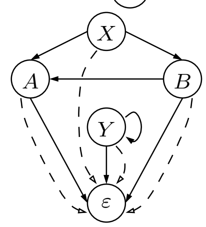
x	$f(x)$	$h(x)$
X	A	XA
Y	A	YA
A	A	A
B	B	B
S	$-$	XAA

$$\textbf{Iteration 2:}\quad \begin{aligned} S &= XAA \\ X &= aA + bYAY \\ Y &= aY + bB \\ A &= a \\ B &= aXAA + b \end{aligned}$$



x	$f(x)$	$h(x)$
X	X	X
Y	B	YB
A	A	A
B	B	B
S	$-$	XAA

$$\textbf{Iteration 3:}\quad \begin{aligned} S &= XAA \\ X &= aA + bYBAYB \\ Y &= aY + b \\ A &= a \\ B &= aXAA + b \end{aligned}$$



x	$f(x)$	$h(x)$
X	X	X
Y	Y	Y
A	A	A
B	B	B
S	$-$	XAA

Theorem 3. *The algorithm* $PNF(G, S)$ *correctly computes a PNF simple grammar* G' *and* S' *such that* $L_G(S) = L_{G'}(S')$.

Proof. Step 1 does not change the semantics of any nonterminal, so it reduces G to an equivalent simple grammar. Step 2 effectively finds final primes for all nonterminals. Step 3 transforms the grammar G into G' by right-factorizing every non-prime nonterminal X by $f(X)$: If X is prime then $L_G(X) = L_{G'}(X)$, otherwise $L_G(X) = L_{G'}(Xf(X))$. Every production $(X \to \alpha) \in P$ is rewritten accordingly into a corresponding production $(X \to \beta) \in P'$. Hence, for all $X \in N$, $L_G(X) = L_{G'}(h(X))$. Thus morphism h converts grammar G together with S to a grammar G' with $S' = h(S)$ such that $L_G(S) = L_{G'}(S')$. Every iteration of the program cuts the length of non-prime nonterminals, in terms of their prime decomposition, by one. Thus, the total number of iterations equals the maximum length of the prime decompositions of nonterminals of the initial grammar. Hence the algorithm terminates. By the exit condition from Step 2, each nonterminal is prime, hence G is in PNF. $\square$

Both steps, 2 and 3, of the algorithm may be computed in linear time, hence the complexity of each iteration of the main loop is dominated by grammar reduction from step 1.

The polynomial time algorithm from Section 6, repeated $O(n^2)$ times may be used to perform the grammar reduction. However, for grammar $G = \{(A_1 \to aA_2A_2), (A_2 \to aA_3A_3), \ldots, (A_{n-1} \to aA_nA_n), (A_n \to a)\}$, language $L_G(A_1)$ has an exponential number of primes with respect to the size of G. Hence the number of iterations of the main loop of $PNF(G, S)$ may be exponential and so may be the size of the resulting PNF grammar. Since simple languages constitute a free monoid, the PNF form is unique.

Corollary 1. *Every simple language L can be represented by a PNF simple grammar $G = (\Sigma, N, P)$ and a starting word $S \in N^*$, such that $L_G(S) = L$. Such a representation is unique. The problem of constructing the PNF representation of L given by a simple grammar is decidable. The PNF representation may be of exponential size with respect to the size of the original grammar.*

4 First mismatch-pair problem

Our approach to transform Caucal's algorithm for the equivalence problem of simple grammar, cf. [1], into a polynomial time one (with respect to the size of the input grammar) is to use compressed representations of sequences of nonterminals, instead of using explicit representations.

We will use the terminology of *acyclic morphisms* because it is more convenient in presenting our algorithms. It is basically equivalent to the representation of a single word by a context-free grammar generating exactly one word, or to a "straight line program".

A *morphism* over a monoid M is an application $H : M \mapsto M$ such that $H(1_M) = 1_M$ and $H(x \cdot y) = H(x) \cdot H(y)$, for all $x, y \in M$. A morphism $H : M \mapsto M$ is fully defined by providing the values for the generators of M. Thus, a morphism H over a finitely generated free monoid N^* is usually defined by providing $H : N \mapsto N^*$. A morphism $H : N \to N^+$ is said to be *acyclic* if we can order elements of N in such a way that for each $A \in N$, we have: $H(A) = A$ or $A > B$ for each symbol B occurring in the string $H(A)$. For an acyclic morphism H over N^* we denote $H^{|N|}$ by H^*, since $H^{|N|+1} = H^{|N|}$. If $H^*(\alpha) = w$ then we say that (H, α) is a compressed representation of w. The size of w can be exponential with respect to the size of its compressed representation.

Let $G = (\Sigma, N, P)$ be a simple grammar. We say that an acyclic morphism $H : N \mapsto N^+$ is *self-proving* in G if for each $A \in N$ we have:

- If $A \to a\,\alpha$ then $H(A) \to a\,\beta$ and $H^*(\alpha) = H^*(\beta)$; and
- If $H(A) \to a\,\beta$ then $A \to a\,\alpha$ and $H^*(\alpha) = H^*(\beta)$.

The concept of self-proving relations was introduced by Courcelle, c.f [3]. The idea of Courcelle and the following Lemma are reformulated in the terms of acyclic morphisms and given here for completeness.

Lemma 3.
If H is an acyclic morphism self-proving in $G = (\Sigma, N, P)$, then $L_G(x) = L_G(H(x))$, for every $x \in N^*$.

The crucial tool in the polynomial-time algorithms is the compressed *first mismatch-pair* problem, *First-MP*:

Input: an acyclic morphism $H : N \mapsto N^+$ and two strings $x, y \in N^*$;

Output:
- *First-MP*(x, y, H) = *nil*, if $H^*(x) = H^*(y)$;
- *First-MP*(x, y, H) = *failure*, if one of $H^*(x)$, $H^*(y)$ is a proper prefix of the other;
- *First-MP*$(x, y, H) = (A, B) \in N \times N$, where (A, B) is the *first mismatch pair*, i.e., the first symbols occurring at the same position in $H^*(x)$ and in $H^*(y)$, respectively, which are different.

We say that a morphism H is binary if $|H(A)| \leq 2$ for each $A \in N$. The following fact can be shown using the algorithm from [13].

Lemma 4. *Assume that given acyclic morphism $H : N \mapsto N^+$ is binary and that the length of x and y is at most $O(|N|)$, then we can solve First-MP(x, y, H) in time $O(k^2 \cdot h^2)$, where $k = |N|$ and $h \stackrel{\mathtt{def}}{=} \min\{k \geq 0 \mid H^k = H^{k+1}\}$ is the depth of the morphism.*

5 The equivalence algorithm

Conceptually it is easier to deal with grammars in binary Greibach Normal Form (denoted GNF2). This means that each side of the production is of the form $(A \to a\alpha)$, where $a \in \Sigma$ and $\alpha \in \{\epsilon\} \cup N \cup N^2$.

Lemma 5. *For each simple grammar G of total size n (the total number of symbols describing G) there is an equivalent simple grammar G' in GNF2 with only $O(n)$ nonterminal symbols. G' can be constructed from G in $O(n)$ time.*

The total size of a grammar in GNF2 is of a same order as the size of N. Hence by the size of a grammar $G = (\Sigma, N, P)$, we mean $n = |N|$.

All known algorithms for the equivalence problem for simple grammars are based on the possibility of computing the quotient of one prefix language by another, assuming that the quotient exists and the languages are given as two nonterminals of a simple grammar.

More precisely, let A and B be two nonterminals of a simple grammar $G = (\Sigma, N, P)$, such that $L(A) = L(B) \cdot L$, for some language $L \subseteq \Sigma^*$. The language L can be derived from A by a leftmost derivation following any word w from $L(B)$, i.e., $A \xrightarrow{*} w\gamma$, for $\gamma \in N^*$, and $L(\gamma) = L$.

Let $||A||$ denote the length of the shortest word derivable from A.

Lemma 6. *Let G be a simple grammar of size n. We can compute the lengths of shortest words derivable from all nonterminals of G in time $O(n \log n)$.*

Proof. Finding $||A||$ for all $A \in N$ corresponds to the single-source shortest paths problem in an *and/or* graph, which, using Dijkstra algorithm, can be solved in time $O(n \log n)$. □

Lemma 7. *Let A and B be two nonterminals of a simple grammar $G = (\Sigma, N, P)$ such that $L(A) = L(B) \cdot L$, for some $L \subseteq \Sigma^*$. We can compute $\gamma \in N^*$ such that $L(\gamma) = L$ in time $O(n)$. It is guaranteed that $|\gamma| \leq n$.*

Proof. Consider the parse tree for the derivation of a shortest word w from A. The idea is to find a path down the tree which cuts off left of this path subtrees γ generating prefix of w of length $||B||$. Since w is a shortest word, no path of the parse tree contains two occurrences of the same nonterminal hence the depth of the tree is at most n. Therefore $|\gamma| \leq n$ and computing the value takes $O(n)$ time. □

The result of the algorithm for finding the quotient of A by B as described in the proof of Lemma 7 will be denoted by $quot(A, B)$. The algorithm will give a result for any pair of nonterminals A and B, as long as $||A|| \geq ||B||$. Notice that $L(A) = L(B\, quot(A, B))$ only if $L(B)$ is a left divider of $L(A)$.

Lemma 3 is the starting point for the design of the algorithm *EQUIVALENCE*. Assume that we fix a linear order $A_1 < A_2 \ldots < A_n$ of nonterminals, such that whenever $i < j$, we have $||A_i|| \leq ||A_j||$. The idea of the algorithm is to construct a self-proving morphism H or, in the process of its construction, to discover a failure which contradicts $L(A) = L(B)$. The main point of the algorithm is to keep pairs of long strings in compressed form. We keep only strings of linear length, their explicit representations are determined by the morphism H. Each time a new rule is generated by setting $H(A) = B\gamma$, where $\gamma = quot(A, B)$, we create pairs (α, β) such that $A \to a\,\alpha$ and $B\gamma \to a\,\beta$, for every letter a of the terminal alphabet. We keep the generated pairs in set Q. To each pair we apply operation *First-MP*, which "eliminates" the next nonterminal, or finds that we have a pair of identical strings, such pairs are removed from Q. By doing that, the algorithm is checking locally for the proof of the nonequivalence of A and B. If the nonequivalence is not discovered and there is nothing to process, i.e., Q is empty, the algorithm returns the value TRUE, meaning $L(A) = L(B)$.

The algorithm *EQUIVALENCE* is presented in Fig. 2. For technical reasons (to simplify the description of the algorithm) we assume that *First-MP*(x, y, H) gives ordered pairs in the sense that if *First-MP*$(x, y, H) = (A, B)$ then $A > B$. For $\alpha \in N^+$ and $a \in \Sigma$, by $\alpha \xrightarrow{a}$ we denote that there is a $\beta \in N^*$ such that $\alpha \to a\,\beta$, and by $\alpha \stackrel{a}{\not\to}$ that there is not. We write $(\alpha, \beta) \xrightarrow{a} (\alpha', \beta')$ to say that $\alpha \to a\,\alpha'$ and $\beta \to a\,\beta'$.

Lemma 8. *The algorithm is correct. The algorithm makes $O(n)$ iterations.*

Proof. In each iteration, either a pair of strings is removed from Q, or a nonterminal is "eliminated" and no more than $|\Sigma|$ pairs of strings are inserted into Q. The crucial property is that whenever $H(A) = B\gamma$, then the nonterminals in $B\gamma$ are of smaller rank than A, ensuring that H is acyclic. Note also that *First-MP* returns a pair (A, B) such that $H(A) = A$, therefore a nonterminal can only be "eliminated" once. After at most $n - 1$ eliminations, *First-MP* will either find that $H^*(\beta_1) = H^*(\beta_2)$ and remove the pair from Q or return $failure$. Thus, the maximum number of iterations is $O(n)$.

Correctness follows from Lemma 3. □

Corollary 2. *The algorithm EQUIVALENCE(X, Y, G) works in time $O(n\, F(n))$, where n is the size of G, and $F(n)$ is the complexity of the First Mismatch-Pair Problem.*

Input: Simple grammar $G = (\Sigma, N, P)$ and nonterminals $X, Y \in N$;
Output: TRUE if $L_G(X) = L_G(Y)$, FALSE otherwise.

```
Initialization:
    Q := {(X,Y)};
    for each A ∈ N do H[A] := A;
while Q is not empty do
    (β1, β2) := an element of Q;
    switch (First-MP(β1, β2, H)) do
        case nil : remove (β1, β2) from Q;
        case failure : return FALSE;
        case (A, B) :
            γ := quot(A, B);
            H[A] := Bγ;        /* The nonterminal A is "eliminated" */
            for each a ∈ Σ do
                if (A, Bγ) -a-> (β1, β2) then insert (β1, β2) into Q;
                if (A -a-> and B -a-/->) or (A -a-/-> and B -a->) then return FALSE;
return TRUE;
```

Fig. 2. Algorithm *EQUIVALENCE*(X, Y, G).

Lemma 9. *Every instance of First-MP*(α, β, H) *in EQUIVALENCE*(X, Y, G) *can be solved in time:*

1. $O(n^6 \log^2 n)$ *and*
2. $O(n^4 \operatorname{polylog} v(G))$.

where n is the size of G, and $v(G)$ is the length of a shortest string derivable from a nonterminal, maximized over all nonterminals.

Proof. In the proof we use twice Lemma 4.

1. Assume H is an acyclic morphism over N, where $n = |N|$ such that $|H(A)| \leq n$ for each A. Then we can construct a morphism H_b such that $H_b^* = H^*$, over a set of $k \leq n^2$ nonterminals and with depth $h = O(n \log n)$.
The transformation of the morphism can be done similarly to a balanced transformation into a Chomsky normal form. If $H(A) = B_1 B_2 \ldots B_n$ then we introduce $n - 2$ new auxiliary nonterminals to change it into a balanced binary tree generating $B_1 B_2 \ldots B_n$ from A. We need $O(n)$ new nonterminals per each original one, altogether the number of nonterminals increases to $O(n^2)$, i.e., k is in $O(n^2)$. However the depth is changed only logarithmically. Observe that on each top down path in generation we have at most n original nonterminals, all of them should be different, and at most $O(n \log n)$ auxiliary nonterminals, i.e., h is in $O(n \log n)$. Now, point 1 follows from Lemma 4.
2. We can use the technique from [16] which transforms each grammar generating a single word u into a grammar of depth $O(\log |u|)$ by introducing $O(n \operatorname{polylog} n)$ new nonterminals. Then Lemma 4 can be applied. □

The series of lemmas gives directly the following theorem, due to the fact that after binarization of the morphism the number of variables grows quadratically and the depth only grows by a logarithmic factor.

Theorem 4. *The algorithm EQUIVALENCE*(X, Y, G) *deciding on the equivalence of two nonterminals* X *and* Y *in a simple grammar* G*, works in time* $O(n^7 \log^2 n)$ *and* $O(n^5 \operatorname{polylog} v(G))$*, where* n *is the size of* G*, and* $v(G)$ *is the length of a shortest string derivable from a nonterminal, maximized over all nonterminals.*

6 Randomized algorithm for *First-MP*

We reduce equality of two compressed texts $H^*(A)$ and $H^*(B)$, to equality of two polynomials of degree at most $\max(|H^*(A)|, |H^*(B)|)$. It is essential that the uncompressed lengths of strings $H^*(A)$ and $H^*(B)$ is only singly exponential. It follows from the construction of the operation *quot* which involves only shortest strings derivable from nonterminals of the grammar G.

Lemma 10 (Randomized Equality Testing). *We can check if* $H^*(A) = H^*(B)$ *in* $O(n \operatorname{polylog} n)$ *randomized time.*

Lemma 11. *The first mismatch-pair problem can be solved by a randomized algorithm in time* $O(n^2 \operatorname{polylog} n)$*.*

Proof. We can check the equality of two prefixes of $H^*(A)$ and $H^*(B)$ at the same time as the equality of $H^*(A)$ and $H^*(B)$. This can be done by changing H into H' which generates only corresponding segments. We omit the details. Then Lemma 10 can be applied. If we can compute the equality of prefixes then we can do a binary search to compute the first mismatch. We have to add as a coefficient the number of iterations in the binary search. This number is logarithmic with respect to the lengths of uncompressed strings, hence it is $O(n)$, since the lengths are only singly exponential. This completes the proof. □

Theorem 5. *We can solve the equivalence problem for simple grammars by a randomized algorithm in* $O(n^4 \operatorname{polylog} n)$ *time.*

7 Conclusion

We have given an algorithm converting any simple grammar to its canonical representation called Prime Normal Form. We also improved the complexity of the best existing algorithm verifying equivalence of simple languages. This result may be used to reduce simple grammars, which is the most expensive step of the PNF algorithm. Despite this improvement, this algorithm still works in exponential time in the worst case, since its output may be of exponential size. However, this theoretical limitation does not seem to occur in practice in the context of network packet filtering and classification.

One interesting open problem is to propose a canonical representation of a simple grammar, and an algorithm computing it, such that the size of this representation is polynomial in the size of the original grammar.

References

1. Didier Caucal. A fast algorithm to decide on simple grammars equivalence. In *Optimal Algorithms*, volume 401 of *LNCS*, pages 66–85. Springer, 1989.
2. Bruno Courcelle. Une forme canonique pour les grammaires simples deterministes. *RAIRO informatique*, pages 19–36, 1974.
3. Bruno Courcelle. An axiomatic approach to the Korenjak-Hopcroft algorithms. *Mathematical Systems Theory*, 16(3):191–231, 1983.
4. Jurek Czyzowicz, Wojciech Fraczak, Andrzej Pelc, and Wojciech Rytter. Prime decompositions of regular prefix codes. In *Implementation and Application of Automata, CIAA 2002*, volume 2608 of *LNCS*, pages 85–94. Springer, Tours, France, July 2003.
5. Wojciech Debski and Wojciech Fraczak. Concatenation state machines and simple functions. In *Implementation and Application of Automata, CIAA'04*, volume 3317 of *LNCS*, pages 113–124. Springer, 2004.
6. Wojciech Fraczak and Anna Podolak. A characterization of s-languages. *Information Processing Letters*, 89(2):65–70, 2004.
7. Loukas Georgiadis and Robert Endre Tarjan. Finding dominators revisited: extended abstract. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pages 869–878. SIAM, 2004.
8. M.A. Harrison. *Introduction to formal language theory.* Addison Wesley, 1978.
9. Yoram Hirshfeld, Mark Jerrum, and Faron Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1–2):143–159, 1996.
10. Oscar H. Ibarra and Shlomo Moran. Probabilistic algorithms for deciding equivalence of straight-line programs. *J. ACM*, 30(1):217–228, 1983.
11. M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, number 937, pages 205–214, Espoo, Finland, 1995. Springer-Verlag, Berlin.
12. A. J. Korenjak and J. E. Hopcroft. Simple deterministic languages. In *Proc. IEEE 7th Annual Symposium on Switching and Automata Theory*, IEEE Symposium on Foundations of Computer Science, pages 36–46, 1966.
13. Masamichi Miyazaki, Ayumi Shinohara, and Masayuki Takeda. An improved pattern matching for strings in terms of straight-line programs. *Journal of Discrete Algorithms*, 1(1):187–204, 2000.
14. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
15. Wojciech Plandowski. Testing equivalence of morphisms on context-free languages. In *ESA*, volume 855 of *LNCS*, pages 460–470. Springer, 1994.
16. Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.
17. J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.