International Journal of Foundations of Computer Science

REDUCING SIMPLE GRAMMARS: EXPONENTIAL AGAINST HIGHLY-POLYNOMIAL TIME IN PRACTICE

Cédric Bastien
Dépt d'informatique, Université du Québec en Outaouais, Gatineau PQ, Canada

Jurek Czyzowicz
Dépt d'informatique, Université du Québec en Outaouais, Gatineau PQ, Canada

Wojciech Fraczak
IDT Canada Inc., Ottawa ON, Canada and
Dépt d'informatique, Université du Québec en Outaouais, Gatineau PQ, Canada

Wojciech Rytter
Instytut Informatyki, Universytet Warszawski, Warsaw, Poland

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

The simple grammar reduction is an important component in the implementation of Concatenation State Machines (a hardware version of stateless push-down automata designed for wire-speed network packet classification). We present a comparison and experimental analysis of the best-known algorithms for grammar reduction. There are two approaches to this problem: one processing compressed strings without decompression and another one which processes strings explicitly. It turns out that the second approach is more efficient in the considered practical scenario despite having worst-case exponential time complexity (while the first one is polynomial). The study has been conducted in the context of network packet classification, where simple grammars are used for representing the classification policies.

1. Introduction

Simple grammars are a subclass of context-free grammars, which have been introduced by [7]. They lead to efficient parsing by means of single-state deterministic push-down automata. On the other hand, since strings may be represented more efficiently by means of implicit representation of one-word context-free grammar (cf. [9]), the equivalence of languages defined by simple grammars may be viewed as a generalization of equality testing of two grammar-compressed strings. The theoretical background of the algorithms involved is an interesting mixture of string matching, text compression, algebraic theory of processes, and formal language theory.

The simple grammar equivalence problem 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 [7, 5], 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 polynomial time in n and $v(G)$, see [3]. The parameter n is the size of the simple grammar (i.e., the length of the total description of the grammar) and $v(G)$ is the length of a shortest string derived from a non-terminal, maximized over all non-terminals. 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 [6]. A recent paper [1] presented a variation of Caucal's algorithm using a technique developed in the context of pattern matching on compressed strings [8, 9], improving the time complexity of the algorithm to $O(n^7 \log^2 n)$ and $O(n^5\ polylog\ v(G)))$. Interestingly, this seemingly theoretical problem, for which a polynomial-time algorithm was unknown for many years, turns out to have an important application in the domain of network packet processing. Moreover, in spite of their high worst-case complexities, the structure of these algorithms makes them potentially applicable in practical situations.

In this paper, we describe our experimental study comparing the performance of three implementations of simple grammar reduction in the context of the wire-speed network packet classification problem. These three implementations are variations of a general simple grammar reduction method but differ by using three different algorithms for deciding on simple grammar equivalence.

IDT Canada designs computer chips which are responsible for data packet classification at wire speed. In the IDT solution, classes of network packets are represented using simple grammars, and their recognition is made by a so-called Concatenation State Machine [4], a hardware implementation of a single-state push-down automaton. In order to store large sets of classification policies in memory, it is essential to reuse their common parts. A natural way to do this consists in decomposing simple languages into primes (languages not representable as concatenation of two simple languages), 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. Representation of finite automata by Concatenation State Machines can be seen as a compression technique. Indeed, the size of a finite state automaton is sometimes exponentially larger than the size of an equivalent Concatenation State Machine. However, certain problems which are easy for finite state machines, like language equivalence (by automata minimization) are much more complex for Concatenation State Machines, as evidenced by this paper. Despite the fact that a Concatenation State Machine is a compact representation of an automaton, in practical cases the sizes of Concatenation State Machines are still large (several tens of thousands of nodes). Hence, the complexities of the algorithms involved are of fundamental importance. Our experiments showed that despite a very large worst-case complexities of the considered algorithms, in practice some of them performed well.

2. Simple grammar equivalence algorithms

A context-free grammar $G = (\Sigma, N, P)$ is composed of a finite set Σ of *terminals*, a finite set N of *non-terminals* disjoint from Σ, and a finite set $P \subset N \times (N \cup \Sigma)^*$ of *production rules*. Note that, contrary to the customary way of defining a grammar, no non-terminal is designed as its axiom. 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 non-terminals $\alpha \in N^*$ of a grammar $G = (\Sigma, N, P)$, the *language* derivable from α, denoted $L_G(\alpha)$, is the set of terminal strings derivable from α, i.e., $L_G(\alpha) \stackrel{\texttt{def}}{=} \{w \in \Sigma^* \mid \alpha \xrightarrow{*} w\}$. 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 $(A, \alpha) \in P$, we have $\alpha \in \Sigma N^*$. A grammar $G = (\Sigma, N, P)$ is a *simple grammar* if G is a Greibach Normal Form grammar such that whenever for some $a \in \Sigma$ $(A, a\alpha_1) \in P$ and $(A, a\alpha_2) \in P$, then $\alpha_1 = \alpha_2$. A language is a *simple language* if it can be derived from a simple grammar. Let $G = (\Sigma, N, P)$ be a simple grammar and $\alpha, \beta \in N^*$ two strings of non-terminals. The equivalence problem consists in deciding whether $L(\alpha) = L(\beta)$, also denoted by $\alpha \equiv \beta$. A mapping $H : N \to N^+$ is called a *decomposing morphism* if we can order the elements of N in such a way that for each $A \in N$, $H(A) = A$ or $A > B$ for each symbol B occurring in the string $H(A)$. We can extend this definition to the domain N^* by defining $H(A\alpha) = H(A) \cdot H(\alpha)$, with $\alpha \in N^*$. We denote $H^{|N|}$ by H^* since $H^{|N|+1} = H^{|N|}$. The height of a decomposing morphism H, denoted by $height(H)$, is defined as $\min\{k \geq 0 \mid H^k = H^{k+1}\}$. By $H_{[A \mapsto \alpha]}$ we denote a new mapping $N \to N^+$ which is identical on all non-terminals but A, and $H_{[A \mapsto \alpha]}(A) \stackrel{\texttt{def}}{=} \alpha$.

Let $G = (\Sigma, N, P)$ be a simple grammar. A decomposing morphism H is said to be self-proving in G, if for each $A \in N$ we have:

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

It has been proved (e.g., in [1]) that if H is a decomposing morphism self-proving in G, then for every $\alpha \in N^+$ we have $\alpha \equiv H(\alpha)$.

Therefore, if two strings α and $\beta \in N^*$ have the same decomposition, i.e., $H^*(\alpha) = H^*(\beta)$, and H is self-proving, then $\alpha \equiv \beta$. In order to prove that $\alpha \equiv \beta$, it is sufficient to find a self-proving decomposing morphism H, such that $H^*(\alpha) = H^*(\beta)$.

The quotient of A by B, denoted $\texttt{quot}(A, B)$, is a word $\gamma \in N^*$, such that, if it exists, $L(A) = L(B)L(\gamma)$. As shown in [6], using the notion of $||A||$, the *norm* of A, i.e., the length of a shortest word of $L(A)$, it follows that there exists such a γ of length in $O(n^2)$, and it can be computed in time $O(n^2)$, where n is the size of the grammar. This technique of calculating $\texttt{quot}(A, B)$, was not originally considered in [3].

First Mismatch-Pair problem (First-MP) is defined as follows:

Input: decomposing morphism $H : N \mapsto N^+$ and strings $\alpha, \beta \in N^+$;

Output:

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

In the context of the simple grammar equivalence problem, the First-MP problem is important in the application of a process called *DecompositionProcess*.

Input: decomposing morphism $H : N \mapsto N^+$ and strings $\alpha, \beta \in N^+$;

Output:

- If *First-MP*$(\alpha, \beta, H) = nil$, then $DecompositionProcess(\alpha, \beta, H) =$ *success*;
- If *First-MP*$(\alpha, \beta, H) = failure$, then $DecompositionProcess(\alpha, \beta, H) =$ *failure*;
- If *First-MP*$(\alpha, \beta, H) = (A, B)$, then, assuming $||A|| \geq ||B||$, the answer is given by a recursive call to *DecompositionProcess*$(\alpha, \beta, H_{[A \mapsto B \cdot \texttt{quot}(A,B)]})$.

Essentially, this process tries to make $H^*(\alpha)$ and $H^*(\beta)$ equal by updating H with a new decomposition whenever a mismatching pair (A, B) is found. The new decomposition is chosen by supposing that $L(B)$ is a left divider of $L(A)$, i.e., by setting $H_{[A \mapsto B \cdot \texttt{quot}(A,B)]}$, which eliminates the mismatch. This operation is done repeatedly until *First-MP* returns *success* or *failure*, which is bound to occur within $|N|$ steps. The decomposition process constructs a self-proving decomposing morphism, as implied by its definition, which would prove the equivalence of the two input strings.

3. Comparison of the algorithms

We consider three simple grammar equivalence algorithms, which were presented in [3], [6], and [1]. Even though they all use a similar basic idea derived from [7], the manner in which this idea is applied differs significantly in two specific ways.

3.1. Two basic strategies in the algorithms

The first difference exists in the way the self-proving decomposing morphism is created.

Incremental algorithms: Both algorithms from [3] and [1] build the decomposing morphism as needed from the input pair of strings. Let S be a set of pairs of non-terminal strings. Initially, S contains only the input pair (α_1, β_1) and H is initialized to $H(A) = A$, for each $A \in N$. The decomposition process is applied to each pair

contained in S. During this process, each time a non-terminal A is assigned a new decomposition in H, the algorithm verifies whether A and $H(A)$ have transitions over the same terminal symbols. If this is not the case, we have failed in building a self-proving decomposing morphism such that $H^*(\alpha_1) = H^*(\beta_1)$ and we conclude that the input strings are not equivalent. Otherwise, for each terminal symbol a for which $(A, a\alpha) \in P$ and $H(A) \to a\beta$, the pair (α, β) is added to the set S. When all elements in S have been processed, we conclude that H is self-proving and, since it has been applied successfully to the input pair of strings, that $\alpha_1 \equiv \beta_1$.

This method of constructing the self-proving decomposing morphism requires only $O(|N|)$ calls to First-MP, which is the only complex operation involved. However, this method does not directly perform grammar reduction; it only determines the equivalence between two non-terminals. In order to obtain a reduced grammar, we call the equivalence algorithm repeatedly over all pairs of non-terminals, increasing the overall complexity by a factor of $O(|N|^2)$.

Decremental algorithm: The algorithm from [6] uses a different method to create the self-proving decomposing morphism. At first, all possible decompositions are considered. That is, for every pair $(A, B) \in N \times N$ such that $||A|| \geq ||B||$, we compute the pair $(A, B \cdot \texttt{quot}(A, B))$. All these potential decompositions are stored in a set S. The objective now is to transform S by removing invalid decompositions, until we have a maximal set of valid decompositions which permits the construction of any self-proving decomposing morphism. In order to do so, we consider every pair $(A, B \cdot \texttt{quot}(A, B)) \in S$ and we verify whether it respects the conditions of the definition of a self-proving decomposition. This is done as before by verifying that A and $B \cdot \texttt{quot}(A, B)$ have transitions for the same terminal symbols. If this is not the case, the pair is removed from S and we continue with the remaining elements in S. Otherwise, for each terminal symbol a where $(A, a\alpha) \in P$ and $B\texttt{quot}(A, B) \to a\beta$, we apply the decomposition process to (α, β), each time with H initialized to $H(A) = A$, for every $A \in N$. Whenever a new decomposition needs to be set, that is whenever a mismatching pair (A', B') is found, we look for its corresponding decomposition $(A', B' \cdot \texttt{quot}(A', B'))$ in S. If a decomposition is found, we set $H_{[A' \to B' \cdot \texttt{quot}(A', B')]}$ and the decomposition process continues. Otherwise, we conclude that the pair $(A, B \cdot \texttt{quot}(A, B))$ cannot be part of a self-proving decomposing morphism and it is removed from S. If any pair is removed from S, we start a new iteration to test all the remaining pairs of S again. If no element can be removed from S, then the process stops. At this point, for every pair $(A, \beta) \in S$ we have $A \equiv \beta$. In particular, for every pair $(A, B) \in N^2$ such that $A \equiv B$, we have $(A, B) \in S$.

3.2. Solving the First-MP Problem — compressed or uncompressed representations

The second difference between the three algorithms exists in the way the First-MP operation is executed. The algorithm from [3] performs this operation directly on uncompressed strings, explicitly decomposing the strings and comparing them symbol by symbol. Since a decomposed string can have an exponential length with respect to the number of non-terminals, the algorithm has exponential complexity.

However, if the lengths of decomposed strings are relatively small, which seems to be the case for all the "real-life" examples we have considered, this approach may be acceptable in practice.

In contrast, the algorithms from [1] and [6] process the First-MP operation in polynomial time without a complete decompression of the strings by using a dynamic programming approach.

Lemma 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 "binary" morphism H_b over a set $N' \supseteq N$ of at most n^2 non-terminals such that:*

- $|H_b(X)| \leq 2$ *for all* $X \in N'$;
- $H_b^*(A) = H^*(A)$ *for all* $A \in N$; *and*
- H_b *is of height* $O(n \log n)$.

Let H be a binary acyclic morphism over N. Denote by *First-GMP*(A, B) the first position in $H^*(A)$ which contains a symbol different from the corresponding symbol in $H^*(B)$. The basic data structure needed to compute *First-GMP* is the table of *overlapping occurrences*, where by an occurrence of a string we mean its starting position. Assume we have a rule $H(A) = BC$. The *splitting point* of X is the position between $H^*(B)$ and $H^*(C)$ in $H^*(X)$, i.e., the last occurrence of $H^*(C)$ in $H^*(X)$. We define the overlap-occurrences table $\mathcal{P}$, where for each two variables $X, Y \in N$, $\mathcal{P}(Y, X)$ is the set of occurrences of $H^*(Y)$ in $H^*(X)$ overlapping the splitting point of X. The following key property of the sets $\mathcal{P}(Y, X)$ follows from the so called *periodicity-lemma*.

Property A: Each set $\mathcal{P}(Y, X)$ is a single arithmetic progression.

Hence the set $\mathcal{P}(Y, X)$ can be of exponential size but it has a small representation (starting point, period, and continuation) and membership query in this set can be answered in constant time.

Lemma 2 *Assume that given acyclic morphism H is binary, then we can solve the First-MP problem in time $O(k^2 \cdot h^2)$, where k is the number of non-terminals and $h = height(H)$ is the height of the morphism.*

The last lemma together with Lemma 1 implies the following theorem (cf. [1]).

Theorem 1 *The First-MP problem for an acyclic morphism can be computed in $O(n^6)$ time.*

4. Implementation

We wrote three programs, `SGR-1`, `SGR-2`, and `SGR-3`, for simple grammar reduction which implement the algorithms from [3], [6], and [1], respectively, for checking on simple grammar equivalence.

The First Mismatch Problem was solved using techniques from the fully compressed string matching, [8]. Besides the First-MP problem, the three programs are relatively simple to implement. The extent to which the implementation can

be improved depends on the structure and the internal mechanisms of the algorithm. Below we list the improvements made in comparison with a straightforward implementation.

Lazy evaluation: Programs `SGR-2` and `SGR-3` compute the First-MP using dynamic programming. However, in case of `SGR-3`, we do not need to consult the set of all entries of the table at each call, but often only a small subset of it. Therefore, we implemented the dynamic programming section of the algorithm using "lazy evaluation", that is we compute any required value of the table only once, the first time it is needed, and store the result in the table for future references. Unnecessary values are never computed.

Reduced number of calls: When performing grammar reduction with `SGR-1` or `SGR-3`, we can reduce the number of calls to function `equivalence`(A, B), which checks for the equivalence of two non-terminals A and B, by sorting the non-terminals according to the length of the shortest word they generate. This permits us to ignore all pairs for which the length of a shortest word derivable from each non-terminal differs.

Reduced redundant calculation of the decomposing morphism: In `SGR-1` and `SGR-3`, whenever two non-terminals are found equivalent, the self-proving decomposing morphism H which was built during the verification of the equivalence, will be reused for subsequent calculations.

Reduced redundant calculation of the table of overlapping occurrences. In `SGR-3` the construction of the dynamic programming table $\mathcal{P}$ (see Section 3.2) depends only on the non-terminals of the grammar and the decomposing morphism used to decompose the symbols. Even though the algorithm introduces new temporary non-terminals before each regeneration of the table, which are needed to *compress* the compared strings of non-terminals before starting the computation of the first mismatch, the original non-terminals used in the morphism are always the same, and they do not depend on the temporary ones. Therefore, if two consecutive regenerations of table $\mathcal{P}$ use the same decomposing morphism, we can reuse the part of the table which is related to the original non-terminals. Whenever a call to `equivalence`() returns *false*, any changes to the table and the decomposing morphism made during that call should be rolled back, if we want to avoid the recalculation of them from scratch. We achieve this by *saving* table $\mathcal{P}$ and the decomposing morphism before each call to `equivalence`().

5. Experimental results

In this section we describe a performance comparison of the three programs `SGR-1`, `SGR-2`, and `SGR-3`. The benchmark of the test-cases on which the experiments were performed came from a real-life example of simple grammars used at

IDT Canada for representing different policies of network packet filtering and classification. We have considered three different classes of simple grammars coming from three different applications. Namely:

(Class A) Every test-case from this class defines a valid HTTP packet over TCP with constraints correlating specific IP source and destination addresses with HTTP headers defined by simple regular expressions.

(Class B) Test-cases from this class describe different policies for Sun content load balancing blades. Such a blade offers Layer 4 through Layer 7 load balancing. The parsing is based on IP protocol and TCP/UDP ports (Layer 4) or URLs, cookies, and CGI scripts (Layer 7).

(Class C) This class contains a set of policies demonstrating the capability of PAX.port (a programmable wire-speed packet classification co-processor) working as a firewall (Layer 3).

All test cases use a binary alphabet. Some other characteristics of the grammars in each class are as follows:

	Class A			Class B			Class C		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Nb. of non-terminals	2878	11784	29520	1852	2568	3478	1122	4555	5765
Nb. of production rules	4972	19025	49735	2418	3415	4944	1707	5254	6789
Avg shortest word length	180	216	317	62	218	280	201	281	307
Max shortest word length	374	622	1064	624	773	824	680	680	680

We compared the performances of the programs by calling each of them over several examples of the three classes of input grammars and measuring the time taken by each one to compute a reduced simple grammar, i.e., a simple grammar equivalent to the input grammar such that no two non-terminals are equivalent. The results, presented in Fig. 1, are compiled separately for each of the three classes of test-cases and sorted by number of non-terminals, each column representing one particular test-case.

Program `SGR-1` performs well for all the test-cases. There is very little variation in the time taken by this algorithm to perform the reduction of different grammars of similar sizes, making it a good practical solution to the problem of simple grammar reduction. Although both programs `SGR-1` and `SGR-3` can handle all test cases, program `SGR-1` usually gives the best results.

Program `SGR-3` theoretically requires $O(n^4)$ memory space. However, the various improvements applied to the implementation of this algorithm, mainly the use of "lazy memory allocation", reduced the amount of memory needed to execute the algorithm over these test-cases.

Program `SGR-2` is ineffective, since it could not compute a result within the allocated time frame, even for the smallest test-cases. In order to estimate the behavior of algorithm `SGR-2`, we generated several simple grammars, varying the number of non-terminals from 10 to 2500. It is important to note that those simple grammars were randomly generated.

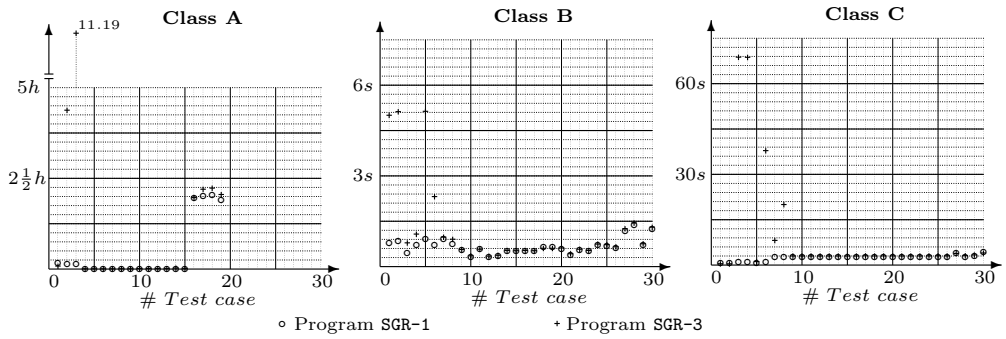

Figure 1: Experimental results for grammar reduction running time. Each column represents a different test case, sorted from left to right by number of non-terminals. Class A grammar reduction running time is expressed in hours and class B and C grammar reduction running times are expressed in seconds.

The table in Fig. 2 illustrates the properties of the generated grammars. Each line describes a particular set of grammars, by means of an arithmetic progression representing the number of non-terminals of each grammar in the set. The column *times* specifies how many different grammars have been tested for each value of the arithmetic progression. The results, presented in Fig. 2, are averaged for each size and sorted by the number of non-terminals of each grammar. It was impractical to test input grammars with more than 2500 non-terminals.

Min	Step	Max	Times
10	10	90	10
100	50	950	10
1000	250	2000	3
2500	− − −	2500	1

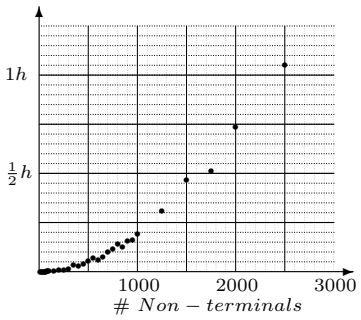

Figure 2: Running time for SGR-2 on randomly generated simple grammars.

6. Conclusion

Despite their high worst case complexities, with a careful implementation, the algorithms for simple grammar equivalence checking from [1] and [3] can both be used to efficiently perform simple grammar reduction, even for large test-cases. The algorithm from [3] in particular has proved to be fast and to have stable performances when applied to different grammars of similar sizes. The good performance of the algorithm from [3] is due to the fact that in all analyzed test-cases the length of the shortest word derivable from a non-terminal was never of exponential size with respect to the size of the grammar. Note, that it is easy to construct a simple grammar of size $O(n)$ generating a single word of length 2^n, for which the algorithm

from [3] is impractical, while the algorithm from [1] runs instantaneously. Since such grammars do not occur in our test-cases, the exponential solution most often yields the best results. Therefore it could be debated whether the additional work needed to implement the polynomial solution to the problem of simple grammar reduction is really worth it, since a much simpler to implement solution based on [3] yields better performance in practice. Note that some part of the good performance of algorithms from [1] and [3] is due to the applied practical improvements. From this perspective the algorithm from [6] was relatively harder to implement.

The algorithm from [6], has proved to be inefficient to solve the problem of simple grammar reduction. Even in the case of grammars consisting of a few hundreds of production rules this algorithm could not produce the result within the allocated time-frame. This is mainly due to the fact that, in the average case, this algorithm performance is relatively close to its theoretical worst-case complexity. However, this algorithm was designed to answer a more general problem than simple grammar reduction, and as such, this result was to be expected. The purpose of [6] was to show that some language theoretical problem has a polynomial-time solution and the authors did not address the question of its optimality. In fact, the context-free processes considered in this algorithm correspond to Greibach Normal Form grammars which are not necessarily deterministic.

It follows from our experiments that the worst-case exponential time algorithm performs in practice like a low-polynomial time algorithm. At the same time, there is a point (which could be called the point of high sophistication of the input) from which the worst-case high-polynomial algorithm substantially beats the exponential algorithm. However, in practical situations the input for our problem does not reach such a high point of sophistication. The practical situations which we considered appear genuinely in the network packet processing applications, and are of great importance in the context of the IDT solution. This analysis made us believe that the two approaches based on algorithms from [3] and [1], may represent a valid practical solution to the problem of simple grammar reduction.

The future work is to implement a *hybrid* algorithm which would combine two categories of algorithms. We can precompute in $O(n \log n)$ time the lengths of the shortest words derivable from all grammar non-terminals. Then, depending on the maximal shortest word v, for example when $v = O(n^2)$, we can run the direct algorithm with explicit decompression of involved strings. Otherwise, the algorithm using sophisticated compressed matching techniques should be used. The practical efficiency of the *hybrid* algorithm would depend on the careful selection of the choice criteria. This requires further work.

Acknowledgments. We would like to thank Feliks Welfeld, Senior Architect at IDT Canada Inc., for providing us with the real-life test examples which made this experimental research possible.

References

1. Cédric Bastien, Jurek Czyzowicz, Wojciech Fraczak, and Wojciech Rytter. Prime

normal form and equivalence of simple grammars. In I. Litovsky J. Farré and S. Schmitz, editors, *CIAA 2005*, volume 3845 of *LNCS*, pages 78–89. Springer, 2006.

2. Cédric Bastien, Jurek Czyzowicz, Wojciech Fraczak, and Wojciech Rytter. Equivalence of Functions Represented by Simple Context-Free Grammars with Output. In *Developments in Language Theory, DLT 2006*, volume 4036 of *LNCS*, pages 71-82. Springer, 2006.
3. Didier Caucal. A fast algorithm to decide on simple grammars equivalence. In *Optimal Algorithms*, volume 401 of *LNCS*, pages 66–85. Springer, 1989.
4. Wojciech Debski and Wojciech Fraczak. Concatenation state machines and simple functions. In *Implementation and Application of Automata, CIAA 2004*, volume 3317 of *LNCS*, pages 113–124. Springer, 2004.
5. M.A. Harrison. *Introduction to formal language theory.* Addison Wesley, 1978.
6. 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.
7. 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.
8. 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.
9. Wojciech Rytter. Grammar Compression, LZ-Encodings, and String Algorithms with Implicit Input. In *ICALP 2004*, volume 3142 of *LNCS*, pages 15–27. Springer, 2004.