Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT

Information Processing Letters 90 (2004) 3–6

Information Processing Letters

www.elsevier.com/locate/ipl

A PTIME-complete matching problem for SLP-compressed words

N. Markey [a,b], Ph. Schnoebelen [b,*]

[a] *Département d'Informatique, Université Libre de Bruxelles, 1050 Brussels, Belgium*
[b] *Laboratoire Spécification et Vérification, ENS de Cachan, CNRS UMR 8643, France*

Received 19 September 2003; received in revised form 2 January 2004

Communicated by L. Boasson

Abstract

SLP-compressed words are words given by simple deterministic grammars called "straight-line programs". We prove that the problem of deciding whether an SLP-compressed word is recognized by an FSA is complete for polynomial-time.

Keywords: Algorithms; Pattern-matching; Compressed strings; Complexity

1. Introduction

Data compression is a powerful and versatile (hence popular) technique for reducing storage space. The issue of finding efficient algorithms working directly on compressed data has received a lot of attention recently. In this field, a paradigmatic family of problems are (searching and) matching problems on compressed strings [10]. This is because strings are the most basic data structure, while searching and matching are ubiquitous problems.

For strings, many compression schemes exist [11]. At a conceptual level, the main compression schemes are SLP (for "Straight-Line Programs"), LZ (for "Lempel–Ziv"), RLZ (for "Restricted Lempel–Ziv") and LZW (for "Lempel–Ziv–Welch"). These four schemes allow *exponential compression*: compressed strings may denote full-length texts of exponential length. They are more or less equivalent since it is possible to translate in polynomial-time compressed strings from one scheme to another. Hence these compression schemes are successful on the same instances, and polynomial-time algorithms for one scheme can be transfered to the others.

Many matching problems on compressed strings can be solved in polynomial-time [9]. But there are situations where polynomial-time is not good enough: since compressed texts can be quite large, it is interesting to have efficient parallel algorithms (i.e., algorithms in NC) or polynomial-time algorithms that only use polylog-space (i.e., in SC)[1].

It has recently been observed that some matching problems on compressed strings admit NC algorithms [5,4,6]. However NC or SC algorithms are not always possible,[2] and some matching problems

* Corresponding author.
E-mail addresses: nmarkey@ulb.ac.be (N. Markey), markey@lsv.ens-cachan.fr (N. Markey), phs@lsv.ens-cachan.fr (Ph. Schnoebelen).

[1] We refer to [8] or [7] for more details on classes below PTIME.

[2] Here and in the following we implicitly make the standard assumption that PTIME does not collapse to NC or SC.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2004.01.002

on compressed strings are known to be complete for polynomial-time [3,4].

When it comes to algorithms in NC or SC, the three main compression schemes (SLP, LZ and LZW) are not equivalent. Regarding complexity below PTIME, the available results are scarce. Some problems are PTIME-hard for LZ [5,9,4] but not for SLP. The evidence seems to indicate that SLP-compressed words are easier (that is, more amenable to efficient algorithms) than (R)LZ(W)-compressed words.

In this note, we prove that deciding whether an SLP-compressed word is accepted by a (fixed) finite-state automaton (an FSA) is PTIME-complete. This is the first example of a PTIME-hard problem for SLP-compressed words.

We also consider matching problems simpler than FSA-acceptance, namely telling whether a given string occurs in the SLP-compressed word. For these problems, we provide simple proofs showing they admit LOGCFL algorithms, hence are low inside NC.

2. Preliminaries

We follow [9]. A *straight-line program*, or SLP, is a context-free grammar where the non-terminals $N_1, \dots, N_m$ are ordered (N_m being the axiom), and where every non-terminal has a single production of the form $N_i \to a$ for a terminal a, or $N_i \to N_j N_k$ for some $j, k < i$. For an SLP P, we write $w(N_i)$ for the unique word described by N_i. Then $w(P)$ stands for $w(N_m)$.

Proposition 2.1 [9]. *Saying whether $w(P)$ is recognized by $\mathcal{A}$ (for P an SLP, and $\mathcal{A}$ a finite-state automaton) can be done in time* $O(|P| \times |\mathcal{A}|^3)$.

Proof. [9] describes a simple dynamical programming solution. For two states r, s of $\mathcal{A}$ and non-terminal N_i of P, set $T[r,s,i] = \mathtt{true}$ iff $w(N_i)$ labels a path going from r to s in $\mathcal{A}$. Obviously, if $N_i \to N_j\ N_k$ is a rule in P, then $T[r,s,i] = \bigvee_u T[r,u,j] \wedge T[u,s,k]$. Hence the table $T[\dots]$ is easy to fill. Then we can use $T[\dots]$ to see whether $w(P)$, i.e., $w(N_m)$, labels an accepting path. □

3. The main result

Theorem 3.1. *Saying whether $w(P)$ is recognized by a deterministic finite-state automaton $\mathcal{A}$, is* PTIME-*complete. Furthermore,* PTIME-*hardness already occurs for a fixed FSA.*

In view of Proposition 2.1, only the second part of Theorem 3.1 has to be proved.

For this, we start by describing $\mathcal{A}_5$, the fixed FSA, and some of its properties. We were inspired by [1] for this construction. $\mathcal{A}_5$ has 5 states numbered from 0 to 4 and is depicted in Fig. 1. The initial state is 0 and the only final state is 1.

For two states $s, t \in \{0, 1, 2, 3, 4\}$, we write $s \xrightarrow{a} t$ ($s \xrightarrow{b} t$, $s \xrightarrow{c} t$) when there is an a-labeled (resp., b-labeled, c-labeled) arrow from s to t. We write $s \xrightarrow{\texttt{inc}} t$ when $t = s + 1 \bmod 5$: hence $\xrightarrow{\texttt{inc}}$ rotates one step clockwise. Observe that the relations $\xrightarrow{a}$, $\xrightarrow{b}$, $\xrightarrow{c}$, and $\xrightarrow{\texttt{inc}}$ are in fact bijections. Further observe that $\xrightarrow{\texttt{inc}} = \xrightarrow{b}\xrightarrow{a}$ (we compose functions from left to right: $f.f'$ denotes $f' \circ f$). Finally, we let *id* denote the identity between states.

As the reader will easily check, the construction of $\mathcal{A}_5$ ensures that the following equalities hold:

$$\xrightarrow{a}\xrightarrow{b}\xrightarrow{c}\xrightarrow{b}\xrightarrow{b} = \xrightarrow{\texttt{inc}},$$
$$\xrightarrow{a}\xrightarrow{b}\xrightarrow{\texttt{inc}}\xrightarrow{c}\xrightarrow{b}\xrightarrow{\texttt{inc}}\xrightarrow{b} = \xrightarrow{\texttt{inc}},$$
$$\xrightarrow{a}\xrightarrow{\texttt{inc}}\xrightarrow{b}\xrightarrow{c}\xrightarrow{\texttt{inc}}\xrightarrow{b}\xrightarrow{b} = \xrightarrow{\texttt{inc}},$$
$$\xrightarrow{a}\xrightarrow{\texttt{inc}}\xrightarrow{b}\xrightarrow{\texttt{inc}}\xrightarrow{c}\xrightarrow{\texttt{inc}}\xrightarrow{b}\xrightarrow{\texttt{inc}}\xrightarrow{b} = id. \quad (1)$$

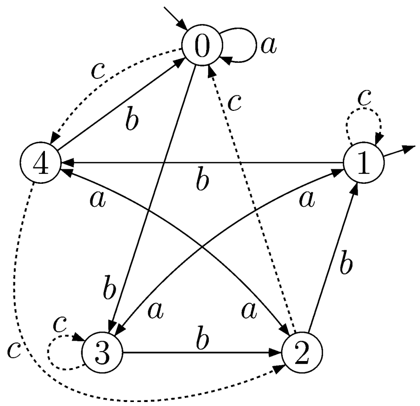

Fig. 1. $\mathcal{A}_5$, a fixed FSA.

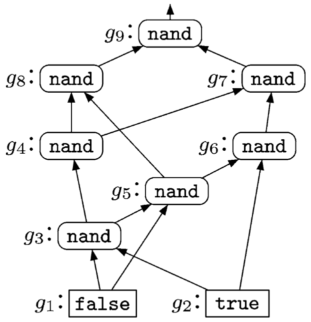

Fig. 2. $\mathcal{C}$, a Boolean circuit of $\mathtt{nand}$ gates.

We now prove Theorem 3.1 by reduction from NAND-CIRCUIT-VALUE. Let $\mathcal{C}$ be a Boolean circuit made of $\mathtt{nand}$ gates with fan-in 2, some constant inputs, and with one designated output gate g. Fig. 2 displays an example. Every gate g_i evaluates to some $v(g_i) \in \{\mathtt{true}, \mathtt{false}\}$ in the obvious way, and we let $v(\mathcal{C})$ denote the value of the output gate. It is well-known that deciding whether $v(\mathcal{C}) = \mathtt{true}$ is a PTIME-complete problem [7, Problem A.1.5].

With a circuit like $\mathcal{C}$, we associate a grammar $P_\mathcal{C}$ that has one non-terminal N_i for every gate g_i. The rule for N_i depends on whether g_i is a $\mathtt{nand}$ gate inputing from g_j and g_k (note that $j = k$ is possible) or it is an input gate carrying $\mathtt{true}$ or $\mathtt{false}$:

if $g_i := \mathtt{nand}(g_j, g_k)$

$$\text{then } N_i \to aN_jbN_kcN_jbN_kb, \tag{2}$$

if $g_i := \mathtt{true}$

$$\text{then } N_i \to ba, \tag{3}$$

if $g_i := \mathtt{false}$

$$\text{then } N_i \to \varepsilon. \tag{4}$$

Finally, if g_m is the output gate in $\mathcal{C}$, then the axiom of $P_\mathcal{C}$ is N_m. (Strictly speaking $P_\mathcal{C}$ is not an SLP, but it is easy to provide an equivalent linear-sized SLP.)

For a word $w = a_1 \ldots a_m$, we write $\xrightarrow{w}$ for the composition $\xrightarrow{a_1} \cdots \xrightarrow{a_m}$ (with $\xrightarrow{\varepsilon}$ being *id*).

Lemma 3.2. *For every gate g_i in $\mathcal{C}$:*

(i) *if $v(g_i) = \mathtt{true}$ then $\xrightarrow{w(N_i)} = \xrightarrow{\mathtt{inc}}$,*
(ii) *if $v(g_i) = \mathtt{false}$ then $\xrightarrow{w(N_i)} = id$.*

Proof. By induction over the height of the gate in the circuit. For the base cases Eqs. (3) and (4) directly ensure (i) and (ii) (recall that $\xrightarrow{ba} = \xrightarrow{\mathtt{inc}}$).

For the inductive step, g_i is some $\mathtt{nand}(g_j, g_k)$. There are four cases. Assume for example that $v(g_j) = \mathtt{true}$ and $v(g_k) = \mathtt{false}$, so that $v(g_i) = \mathtt{nand}(\mathtt{true}, \mathtt{false}) = \mathtt{true}$ and we have to prove $\xrightarrow{w(N_i)} = \xrightarrow{\mathtt{inc}}$. By ind. hyp. $\xrightarrow{w(N_j)} = \xrightarrow{\mathtt{inc}}$ and $\xrightarrow{w(N_k)} = id$. Now Eq. (1) exactly states that $\xrightarrow{w(N_i)} = \xrightarrow{\mathtt{inc}}$, given that N_i is defined by (2). The three other cases use the other equalities in (1). □

Corollary 3.3. $v(\mathcal{C}) = \mathtt{true}$ *iff* $\xrightarrow{w(P_\mathcal{C})} = \mathtt{inc}$ *iff* $w(P_\mathcal{C})$ *is accepted by* $\mathcal{A}_5$.

Thus we have provided a logspace reduction from NAND-CIRCUIT-VALUE to acceptance of SLP compressed words by the fixed FSA $\mathcal{A}_5$, proving the second part of Theorem 3.1. $\mathcal{A}_5$ uses three letters for clarity but a two-letter alphabet would have been sufficient in view of $\xrightarrow{c} = \xrightarrow{bbababab}$.

4. Efficient pattern-matching for SLP-compressed words

In this section we look at pattern-matching problems that are special cases of FSA acceptance, trying to strengthen our PTIME-hardness result by extending it to a problem simpler than FSA acceptance.

It turns out that the problems we consider are in LOGDCFL and LOGCFL respectively, hence they are in AC^1 and admit fast parallel algorithms. Note that these same problems are PTIME-complete for LZ-compressed words [4]. We see this as evidence that SLP-compressed words are "more manageable" than LZ-compressed words even though they are not significantly longer.

We start with the simplest matching problem: does $w(P)$ exactly match a given string p?

Theorem 4.1. *Deciding whether $p = w(P)$ for a word p and an SLP-compressed word P is in* LOGDCFL.

Proof (*Sketch*). We show that the problem can be solved by a deterministic Turing Machine having access to an auxiliary unbounded pushdown storage and working in logarithmic space and polynomial time. Then we conclude relying on the characterization LOGDCFL $=$ **AuxPD-DSPACE**$(\log n, \text{pol}\, n)$ from [2,12].

The algorithm works as follows: Given an SLP P, one puts the axiom N_m on the initially empty stack, and stores a pointer to the beginning of p. The stack will be used to store the sequence of non-terminals that remain to be developed. As long as the stack is not empty, we pop the first non-terminal, N_i (say), from the top of the stack: either P has a rule $N_i \to N_j N_k$, and we add N_k and N_j, in that order, onto the stack; or the rule is $N_i \to a$, and we check that a is the current letter in p, in which case we advance the pointer inside p.

The algorithm runs in time $\mathrm{O}(|p| \cdot |P|)$, and needs space $\mathrm{O}(\log |p| + \log |P|)$ to memorize the pointers inside p and inside P. $\square$

Next we consider *combined patterns* of the form $p_0 \star p_1 \star \cdots \star p_m$ where the p_i are words and where $\star$ means "any substring". Such combined patterns can express problems like "does p occur inside the text?" (by picking $m = 2$, $p_0 = p_m = \varepsilon$ and $p_1 = p$), and "is p a subword of the text?" (for p of the form $a_1 \ldots a_n$, one picks $m = n + 1$, $p_0 = p_m = \varepsilon$ and $p_i = a_i$ for $1 \leqslant i \leqslant n$).

Theorem 4.2. *Deciding whether $w(P)$ matches a combined pattern $p_0 \star \cdots \star p_m$ (for SLP-compressed words P) is in* LOGCFL.

Proof (*Sketch*). We proceed as in the previous theorem, as if we were checking that $w(P)$ equals $p_0 \star \cdots \star p_m$. The difference is that, when we are matching a $\star$ inside the pattern, we just discard letters from $w(P)$. This is achieved by *nondeterministically* guessing the (occurrences of) non-terminals we won't have to develop when we pop them from the stack. Note that this runs in polynomial-time since we can discard a useless (occurrence of a) non-terminal as soon as possible, i.e., before expanding it. (Expanding it and discarding its expansion would require exponential-time.) One concludes with the characterization LOGCFL $=$ **AuxPD-NSPACE**$(\log n, \text{pol}\, n)$ from [2,12]. $\square$

References

[1] M. Beaudry, P. McKenzie, P. Péladeau, D. Thérien, Finite monoids: From word to circuit evaluation, SIAM J. Comput. 26 (1) (1997) 138–152.

[2] S.A. Cook, Characterizations of pushdown machines in terms of time-bounded computers, J. ACM 18 (1) (1971) 4–18.

[3] S. De Agostino, P-complete problems in data compression, Theoret. Comput. Sci. 127 (1) (1994) 181–186.

[4] L. Gąsieniec, A. Gibbons, W. Rytter, Efficiency of fast parallel pattern-searching in highly compressed texts, in: Proc. 24th Internat. Symp. on Mathematical Foundations of Computer Science (MFCS'99), Szklarska Poreba, Poland, September 1999, in: Lecture Notes in Comput. Sci., vol. 1672, Springer, Berlin, 1999, pp. 48–58.

[5] L. Gąsieniec, M. Karpinski, W. Plandowski, W. Rytter, Efficient algorithms for Lempel–Ziv encoding, in: Proc. 5th Scandinavian Workshop on Algorithm Theory (SWAT'96), Reykjavík, Iceland, July 1996, in: Lecture Notes in Comput. Sci., vol. 1097, Springer, Berlin, 1996, pp. 392–403.

[6] L. Gąsieniec, W. Rytter, Almost optimal fully LZW-compressed pattern matching, in: Proc. Data Compression Conference (DCC'99), March 1999, Snowbird, UT, USA, IEEE Comp. Soc. Press, 1999, pp. 316–325.

[7] R. Greenlaw, H.J. Hoover, W.L. Ruzzo, Limits to Parallel Computation: P-Completeness Theory, Oxford Univ. Press, 1995.

[8] D.S. Johnson, A catalog of complexity classes, in: J. van Leeuwen (Ed.), in: Handbook of Theoretical Computer Science, vol. A, Elsevier Science, 1990, pp. 67–161, chapter 2.

[9] W. Plandowski, W. Rytter, Complexity of language recognition problems for compressed words, in: J. Karhumaki, H. Maurer, G. Păun, G. Rozenberg (Eds.), Jewels are Forever, Springer, Berlin, 1999, pp. 262–272.

[10] W. Rytter, Algorithms on compressed strings and arrays, in: Proc. 26th Conf. Current Trends in Theory and Practice of Informatics (SOFSEM'99), Milovy, Czech Republic, November 1999, in: Lecture Notes in Comput. Sci., vol. 1725, Springer, Berlin, 1999, pp. 48–65.

[11] J.A. Storer, T.G. Szymanski, Data compression via textual substitution, J. ACM 29 (4) (1983) 928–951.

[12] I.H. Sudborough, On the tape complexity of deterministic context-free languages, J. ACM 25 (3) (1978) 405–414.