Real-time traversal in grammar-based compressed files*

Leszek Gąsieniec Roman Kolpakov Igor Potapov Paul Sant

Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
{leszek,roman,igor,pauls}@csc.liv.ac.uk

Abstract

We study real-time recovery of consecutive symbols from compressed files, in the context of grammar-based compression, see, e.g., [7]. In this setting, a compressed text is represented as a small (a few kilobytes) dictionary D (containing a set of code words), and a very long (a few megabytes) string based on symbols drawn from the dictionary D. The space efficiency of this kind of compression, is comparable with standard compression methods based on the Lempel-Ziv approach [14]. We show, that one can visit consecutive symbols of the original text, moving from one symbol to another in constant time and extra $O(|D|)$ space. This algorithm is an improvement of the on-line linear (amortised) time algorithm presented in [7].

1 Introduction

Historically, one of the main aims of text compression was to reduce the size of data stored for future analysis and processing. Unfortunately, further processing of the compressed data is usually preceded by complete decompression. This standard approach may cause problems in certain applications, where the original (uncompressed) file is very large and we face running out of space available for the computation. Thus, it is important to be able to process compressed data without requiring (complete) decompression. Moreover, in some applications (e.g., pattern matching problems), the information represented by the compressed file can be processed in relatively small chunks (length of a pattern). In this context it is crucial to study compression methods that allow time/space efficient access to any fragment of a compressed file without being forced to perform complete decompression.

In this paper we are especially interested in real time recovery of consecutive symbols from the compressed (streamed) text. We expect that this type of operation might become useful in enrichment of World Wide Web protocols, such as *Real Time Text Protocol* (RTTP), see [12]. RTTP extends the capability of the Web by providing a mechanism for streaming textual and numeric data with very low latency to any Web-connected device without having to resend an entire document. We also believe that our work has applications in *text stream monitoring* and

*This work is supported by the EPSRC grant GR/R84917/01.

pattern detection used in the context of tracking information flow between terrorists, web parental control, continuous monitoring of rival web sites in e-commerce, and others, see [4].

In [7] we find *linear amortised time* traversal in grammar-based compressed files in the presence of a constant size extra space. In particular, the algorithm allows us to perform constant space compressed string matching in linear amortised time. The approach is based on the observation that one can avoid the use of an extra space if manipulation on a (small) dictionary D (which is stored in the internal memory, or perhaps a processors cache) is permitted. While the traversal algorithm allows us to visit the whole text in linear time, in some parts of the text the time cost of moving between consecutive symbols is potentially as large as $\Omega(|D|)$. We believe that for some applications, such as streaming large textual data, this might be considered unacceptable. Following this line, we propose a real-time traversal algorithm that uses $O(|D|)$ extra space. The extra space is used to store: additional information about the structure of the dictionary D (obtained during the compression process), and a dynamic structure that allows us to keep a trace while traversing through the structure of D. Please note that the compressed text obtained via application of a grammar-based compression used in this paper comprises a small dictionary D which has the form of a *straight line program*, see section 2. The dictionary D is relatively small and it contains several thousand code words that are used to encode the original text in a string usually of size of several millions of code words. In this context, we strongly believe that the use of extra space of size $O(|D|)$ leading to real-time recovery of consecutive symbols in the original file from its compressed version is perfectly justified and promising.

Manipulation of compressed files was recently the subject of intensive studies in the context of the string matching problem, where one is interested in finding all occurrences of a pattern p in a longer text t, see, e.g., [3]. Originally the main focus of the studies was on the time complexity of the search (for pattern occurrences) and most of the algorithms require at least linear extra memory in the size of the whole compressed text. A number of time efficient algorithms were proposed for LZ/LZW-compressed files, see, e.g., [1, 8]. More recently compressed pattern matching was considered in the context of grammar-based compressed files. However, these algorithms are mainly focused on deciding whether a fixed pattern occurs in a text, i.e., the approach does not provide direct access to consecutive symbols of the (uncompressed) text, see, e.g., [11]. Also quite recently Amir *et al.* [2] stated the problem of compressed pattern matching using small extra space. They proposed a time/space efficient pattern matching algorithm that works for the run-length encoding. Their algorithm (designed for 2d-pattern matching) works in linear time and requires only a linear space in the size of a compressed pattern. Their result is feasible due to the fact that the concept of run-length coding is rather simple, however it doesn't identify itself with as good compression ratio as other standard text compression (such as LZ, LZW, or grammar-based) methods do. Another interesting aspect of the compressed search has been studied by Ferragina and Manzini in [5]. They delivered a novel data structure for indexing and searching with a space occupancy being a (linear) function of the entropy of the underlying data sets.

The rest of the paper is organized as follows. In section 2 we recall basic string definitions and we expose more details on grammar-based compression via straight-line programs. In section 3 we show how to recover, in real-time, consecutive symbols of the original text from its fully compressed counterpart. We initially show how to implement a real-time procedure in the presence of $O(|D|^2)$ space. Later we show how to reduce the use of extra memory to $O(|D|)$ via efficient implementation of a *next link* operation defined in the same section. We also propose a mechanism for the real-time recovery of a sequence symbols starting at an arbitrary (though known in advance) position in the uncompressed file. We conclude with a short discussion on possible applications of our new method.

2 Preliminaries

We recall a number of standard definitions used in the context of strings and the grammar-based compression.

Definition 1 *A context free grammar G is a quadruple $\{\Sigma, V, R, S\}$, where*

Σ *is a finite set of symbols called terminals;*

V *is a finite set of symbols disjoint from Σ called variables (or non-terminals);*

R *is a finite set of productions (rewrite rules) $x \to y$, where $x \in V$ and $y \in (\Sigma \cup V)^*$;*

S *is a special symbol called the start symbol (or axiom), $S \in V$.*

Definition 2 *A derivation tree for a string w in a context free grammar $G = \{\Sigma, V, R, S\}$ is a planar tree satisfying:*

– *every vertex has a label, which is a symbol from $V \cup \Sigma$;*

– *the label of the root is S and each internal node has a label from V;*

– *if a vertex has a label A and the $X_1, \ldots, X_k$ are the labels of the immediate descendants of the vertex in from-left-to-right order, then the rule $A \to X_1 \cdots X_k$ must belong to R; we assume that edges leading from A to its children $X_1, \ldots, X_k$ are labelled from 1 to k respectively.*

– *w is the concatenation of labels of the leaf vertices from left to right.*

Definition 3 *A straight-line program (SLP) R is a sequence of assignment statements using variables $X_1, ..., X_u$ and symbols of a given alphabet Σ:*

$$X_1 := expr_1;\ X_2 := expr_2;\ \cdots;\ X_u := expr_u;$$

Each expression $expr_i$ must be either a symbol of a given alphabet Σ or a concatenation of variables $X_{j_1} \cdot X_{j_2} \cdot \cdots \cdot X_{j_k}$, where $j_l < i$ for all $1 < l \leq k$.

Definition 4 *An SLP grammar is a context-free grammar with a set of productions of the form:*

$$X_i \to X_{j_1} \ldots X_{j_k}, \text{ where } j_l < i \text{ for all } 1 < l \leq k.$$

Definition 5 *An SLP tree is a derivation tree of some SLP grammar.*

Definition 6 *An SLP graph is a directed acyclic graph (dag) which is obtained from the SLP tree by merging all nodes labelled by an identical terminal or non-terminal symbol and then by removing all but one multiple (and identical) edges between each pair of new nodes.*

The three representations of SLPs, i.e., SLP grammars, SLP trees and SLP graphs are equivalent. An outline of an SLP grammar for Fibonacci words is shown in Example 1. For an illustration of the SLP tree and the SLP graph for 6th Fibonacci word check Figure 1.

Example 1 *Let us consider the SLP for Fibonacci words:*

$$F(n) \to F(n-1) \cdot F(n-2)$$
$$F(n-1) \to F(n-2) \cdot F(n-3)$$
$$\dots$$
$$F(3) \to F(2) \cdot F(1) \textit{ and } F(1) = a,\ F(2) = b.$$

>From definitions (1 - 6) it follows, that a straight-line program can be represented as an *SLP graph* G, where all nodes of out-degree > 0 stand for non-terminals (new symbols in extended alphabet $\mathcal{A}$) and all nodes of out-degree 0 stand for terminals (symbols in original alphabet A). All productions correspond to non-terminal nodes and their outgoing edges. An SLP can be also expressed as its (usually much larger) SLP tree T with the root labelled by the SLP *starting* non-terminal, see Figure 1. A string generated by the SLP can be obtained by traversing the SLP-tree T and reporting, in left to right order, a content of consecutive leaves in T.

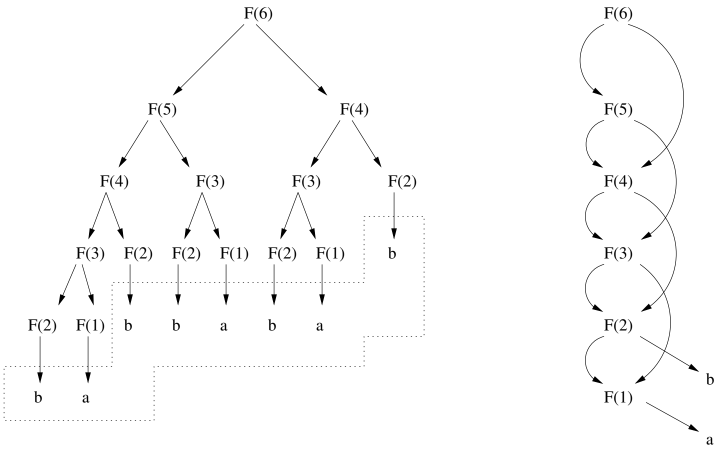

Figure 1: SLP for 6th Fibonacci word. F(6) is a starting non-terminal.

2.1 Grammar-based compression via straight-line programs

There are several compression methods based on a construction of simple deterministic grammars, e.g., see [10]. A crucial idea used here is to find the smallest possible set of productions, i.e., a *dictionary*, that generates the original string or rather a collection of its consecutive segments. In this paper we work on files compressed, e.g, by a *recursive pairing* method [7], where each production always has two elements on its right-hand side. In this context, the compressed representation of the original string is formed by a triplet $(\mathcal{D}, \mathcal{A}, \mathcal{S})$, where: $\mathcal{D}$ denotes the dictionary (set of SLP productions), $\mathcal{A}$ stands for the set of *code words* (all terminals and non-terminals used in SLP productions), and $\mathcal{S}$ is a compressed string built over the extended alphabet $\mathcal{A}$.

Example 2 *A compressed representation of a string* $abbababbabbababbababbabbababbabba$ *is a triplet* $(\mathcal{D}, \mathcal{A}, \mathcal{S})$, *where a dictionary* $\mathcal{D} = \{A \to ab, B \to Ab, C \to BA, D \to CB\}$, *an extended alphabet* $\mathcal{A} = \{a, b\} \cup \{A, B, C, D\}$, *and a compressed string* $\mathcal{S} = DCDDBa$.

3 Real-time consecutive symbols recovery in SLP grammars

In this section we present real-time traversal algorithm in SLPs that can be applied directly to any code word appearing in a compressed text.

In [7] we studied the problem of *sequential* recovery of consecutive text symbols encoded by an SLP in amortised linear time and with the help of extra constant space. The idea of a constant space decompression is based on the traversal through an SLP-tree using formal transformations of a context free grammar. The time associated with moving from one leaf in the SLP tree to its successor during the traversal is proportional to the (tree distance) between the two leaves. Thus this simple traversal method can not guarantee recovery of consecutive symbols (stored in SLP-tree leaves) in real-time. Our aim here is to design a real-time algorithm which reports the content of each consecutive leaf (a symbol in original text) in the SLP-tree with a constant time slowdown.

In what follows we describe the framework of the real-time traversal algorithm for any SLP grammar G. The pseudo-code of the algorithm is presented below. The algorithm starts with the initial symbol S in grammar G and the leftmost terminal symbol t corresponding to the leftmost leaf in the SLP-tree rooted in S. During each round of the algorithm we find the next (to the right) leaf (in the SLP tree), which corresponds to the next symbol in the original text. At the beginning of each round, the algorithm computes the node P (non-terminal symbol in G) which is the *lowest common ancestor* (LCA) of the leaf t and the next, yet *unknown*, leaf in the SLP-tree. Then, it inspects the right child of P (at appropriate SLP-tree node) and if the right child of P is a terminal symbol then next leaf of the SLP tree is found. Otherwise the next leaf of the SLP tree is the leftmost terminal symbol for the right neighbour of P. The algorithm terminates eventually when the LCA non-terminal symbol is not defined.

Procedure 1 Real-time SLP traversal
    $S \leftarrow$ start symbol of SLP grammar
    $t \leftarrow$ the leftmost terminal symbol for non-terminal S in SLP tree
    report(t)
    repeat
        $P \leftarrow$ the LCA for the current and the next *unknown*
            terminal symbol in SLP tree
        $P_{right} \leftarrow$ the right child of P
        if (P_{right} is terminal) **then** $t \leftarrow P_{right}$
        else $t \leftarrow$ the leftmost terminal symbol for a non-terminal P
        report(t is a leaf of SLP tree and a symbol of the original text)
    until (the next LCA is not defined)

In order to report every single symbol of the original text in real-time using this framework we have to provide several functions executable in constant time, including: (1) computation of the leftmost terminal symbol for any non-terminal P, and (2) the LCA for the current and the next *unknown* terminal symbol in an SLP-tree. In what follows we present two real-time traversal algorithms which use $O(|D|^2)$ and $O(|D|)$ extra space respectively. The algorithms have a similar structure, however they differ in the way that the computation of the LCA is performed.

3.1 $O(|D|^2)$ space algorithm

In our algorithm we use the following preprocessed maps L, L_i and H:

- L is mapping $V \to V^* \cdot \Sigma$: L(P)={the sequence of the left derivations for the non-terminal P, which is finished by terminal}
- $L_{\mathbb{N}}$ is mapping $(V, \mathbb{N}) \to V \cup \Sigma$: $L_i(P) = \{$the element on the i-th position in $L(P)\}$
- H is mapping $V \to \mathbb{N}$: $H(P) = |L(P)|$.

In terms of an SLP-tree, $L(D)$ corresponds to the sequence of symbols in the leftmost path from the node D towards the terminal symbol. Since, for any $P \in V$, $H(P) < |D|$, the total size of mapping L is bounded by $O(|D|^2)$. For example, a sequence of left derivations for the non-terminal D (see, Example 2) is $L(D) = [C, B, A, a]$, where $L_1(D) = C$, $L_2(D) = B$, $L_3(D) = A$, $L_4(D) = a$, and $H(D) = |L(D)| = 4$.

The structures based on the mappings provide direct access to all symbols appearing in each sequence of the leftmost derivations, including the leftmost terminal symbol, denoted by $L_{H(P)}(P)$ in a sequence based on non-terminal P. In our solution we will also use a stack R to support efficient computation of the LCA for a terminal symbol t and the next, yet *unknown*, terminal symbol in constant time.

Let R be a stack of pointers to the elements from the sequences of the left derivations for non-terminals and let t be the last visited leaf (terminal symbol) and S be the root in the SLP-tree. The stack R contains a sequence of pointers to some nodes in the SLP-tree that belongs to the path from the root S to a terminal t, at which the path changes its direction from left to right (see Figure 2). We represent each pointer in R as a pair (P, i), where P is a non-terminal and i is an index of the symbol in the sequence $L(P)$.

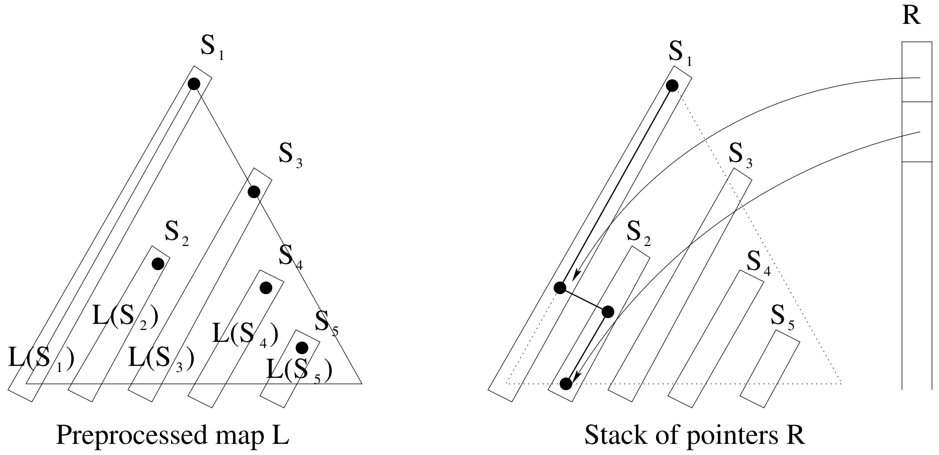

Figure 2: The structures for $O(D^2)$ space algorithm.

We are ready to fill the general framework for the real-time traversal with more details. Initially, we find the leftmost terminal symbol for the root S in its SLP-tree using preprocessed mapping L and *push* $(S, H(S))$ onto the stack R. During each round of the algorithm we compute the LCA for the current and the next, yet *unknown*, terminal symbol using the stack R and the mapping L. In order to find the LCA we *pop* the head element $L_i(F)$ from the stack R and then check its position in the sequence L(F). If $L_i(F)$ is not a child of F (i.e. $i > 1$) then the previous element $L_{i-1}(F)$ in the sequence $L(F)$ is the requested LCA and we *push* it onto the stack R using a direct access to $L_{i-1}(F)$ in the preprocessed mapping L. Otherwise a non-terminal F plays a role of the LCA.

Procedure 2 Quadratic Space

    $S \leftarrow$ start symbol of SLP grammar
    report $L_{H(S)}(S)$
    $R \leftarrow push(S, H(S))$;
    repeat
        — Find the LCA —
        $(F, i) \leftarrow pop(R)$
        if ($i > 1$) then $R \leftarrow push(F, i-1)$; $P \leftarrow L_{i-1}(F)$
        else $P \leftarrow F$
        —– P is the LCA —–
        $P_{right} \leftarrow$ the right child of P
        if (P_{right} is terminal) **then** output (P_{right})
        else report ($L_{H(P_{right})}(P_{right})$)
            $R \leftarrow push(P_{right}, H(P_{right}))$
    until ($|R| = 0$)

Theorem 1 *There exists a real-time $O(|D^2|)$ extra space algorithm recovering consecutive symbols in grammar-based compressed files.*

3.2 $O(|D|)$ space algorithm

The most expensive part (in terms of the use of space) of the algorithm corresponds to the search for the element $L_{i-1}(F)$, while residing in $L_i(F)$, in the sequence of left derivations $L(F)$. In this section we show how to find $L_{i-1}(F)$ in constant time and $O(|D|)$ extra space. Note that the mapping sequence $L(S)$ is uniquely defined for each non-terminal S. In particular, the leftmost terminal symbol appearing in S is also uniquely defined. Thus a set of all non-terminals can be partitioned into classes of abstraction represented by their left-most terminals. The idea of the $O(|D|)$ space real-time recovery of the original text symbols is based on the observation that we can represent all mapping sequences $L(\cdot)$ in $O(|D|)$ space. In order to do so, we define for each sequence $L(S)$ its reversed counterpart $L^R(S)$, in which symbols appear in reverse order. Later we create a number of trees (in fact tries) that are used to represent all sequences $L^R(\cdot)$. The number of trees corresponds to the number of the classes of abstraction represented by the left-most terminals. Note that, since, every non-terminal has only one appearance in one of the trees (associated with the appropriate class of abstraction), the total size of the trees is bounded by $O(|D|)$. However a new representation of $L(\cdot)$ sequences require more sophisticated implementation of the LCA queries. In particular, when we traverse along a sequence $L(F)$, moving from $L_i(F)$ to $L_{i-1}(F)$, in the respective tree we have to be able to solve the *next link* (NL) problem in constant time.

3.2.1 The next link problem

In order to implement a linear space algorithm, we have to resolve the following problem which resembles finding the lowest common ancestor (LCA) of two nodes in a rooted tree, see, e.g., [9]. Let $\mathcal{T}$ be a rooted tree and u, v be two distinct nodes in $\mathcal{T}$, where u is an ancestor of v. The *next link* from u to v is a child of u that is also an ancestor of v. The *next link problem* (NL problem) is to process the tree $\mathcal{T}$, s.t., a next link query for any pair of nodes $(u, v) \in \mathcal{T}$, where u is an ancestor of v, is answered efficiently. In what follows we show a linear (time and space) preprocessing of $\mathcal{T}$ that allows to answer next link queries in constant time.

Our solution to NL problem is based on a variant of the LCA algorithm, due to Schieber and Vishkin, see [9]. While presenting our algorithm we will adopt the notation and definitions used in [9]. In particular, our solution is based on almost identical preprocessing of the tree $\mathcal{T}$. Firstly, we traverse all nodes of the tree $\mathcal{T}$ in a DFS manner, starting at the root of $\mathcal{T}$. During the traversal we label each node v in $\mathcal{T}$ with its prefix number, i.e., its position in order of appearance in the DFS traversal. Simultaneously we also compute for each label k the *height* $h(k)$, i.e., the position of the least-significant bit set to 1 in the binary representation of k, if we count bits from the right. Then, using the algorithm described in [9], for each node v we compute the *run* $I(v)$, which is the number of nodes with the maximal height in the subtree rooted in v (including v itself). Then, for each run k, we compute simultaneously the head $L(k)$, which is the highest node v in $\mathcal{T}$, s.t., $I(v) = k$. Moreover, for each node v we also compute *the main child* $MC(v)$, which is the child v' of v, s.t., $I(v') = I(v)$, if such a child exists. Note that any node v has no more than one main child, and $MC(v)$ can be computed during the computation of $I(v)$. Finally, for each node v we compute its bit mask (represented as a single integer) A_v, which is defined as follows: the ith bit (from the right) in A_v is set to 1 if and only if v has an ancestor u in $\mathcal{T}$, s.t., $h(I(u)) = i$. The computation of A_v is produced in the same way as described in [9].

We now show how to find the next link from node u towards v, where u is an ancestor of v. If $I(u) = I(v)$ then the next link is the main child of u. Assume $I(u) \neq I(v)$. Then we initially find the highest node w in the path from u to v, s.t., $I(w) \neq I(u)$. Note, that $w = L(I(w))$. Thus, in order to compute w we also have to find $I(w)$. Let $d = \lfloor \log n \rfloor + 1$, where n is the number of nodes in $\mathcal{T}$. Any run x in $\mathcal{T}$ can be represented by a bit word consisting of d bits. We will denote this word by $\tilde{x}$. The following fact is shown in [9].

Proposition 1 *Let a node v be an ancestor of a node u, and $h_u = h(I(u))$, $h_v = h(I(v))$, $\tilde{I}(u) = \alpha_d^u \dots \alpha_1^u$, $\tilde{I}(v) = \alpha_d^v \dots \alpha_1^v$. Then $h_u \geq h_v$, and $\alpha_d^u \dots \alpha_{h_u+1}^u = \alpha_d^v \dots \alpha_{h_u+1}^v$.*

Let $h_w = h(I(w))$ and $\tilde{I}(v) = \alpha_d^v \dots \alpha_1^v$. Since w is an ancestor of v, it follows from proposition 1, that:

$$\tilde{I}(w) = \alpha_d^v \dots \alpha_{h_w+1}^v 10 \dots 0. \tag{1}$$

Note also, that u and w are in the path from the root of $\mathcal{T}$ to v, and, that between u and w, there are no nodes with runs different from $I(u)$. Thus, using proposition 1, we conclude that h_w is the position of the left-most 1 in bit-mask A_v that is to the right of position $h(I(u))$. And h_w can be computed from values A_v and $h(I(u))$. Knowing h_w, we can compute $I(w)$ using relation (1) and then we can find $L(I(w))$ whose value is w. It is shown in [9] that all these computations can be implemented in constant time and linear extra space. Consider now the parent $\hat{w}$ of w in $\mathcal{T}$. If $\hat{w} = u$ then w is the next link from u to v. Otherwise the next link is the main child of u.

Theorem 2 *There exists a real-time $O(|D|)$ extra space algorithm recovering consecutive symbols in grammar-based compressed files.*

3.2.2 Reporting text symbols from an arbitrary position

The algorithm presentend in the previous section assumes that the we start the recovery of consecutive text symbols at the beginning of the compressed text. If, for some reason, we would like to start such a recovery process from any other position i, e.g., a compressed text is a collection of smaller documents, we need to enhance our data structures in the following simple way. For every non-terminal symbol X in the dictionary D we also store the length $W(X)$ of the string generated by X. Please note that this update requires only $O(|D|)$ extra space. We also assume here that all potential starting positions are known in advance and that they are stored in the form of an indexed list of *bookmarks*. Each bookmark (for the starting position i) is a pair of numbers (ind, off), where ind is the index of a symbol (possibly non-terminal) in the compressed text containing position i and off is the offset of i in it. While looking for the starting position i we first identify and access the appropriate symbol X in D in constant time. The link to the symbol in D is available in the compressed (random access) file at the position ind. Later, we navigate through the symbols of D in the following way. Being at symbol X, s.t., $W(X) > 1$, we know that X has exactly two out-neighbours (children), i.e., the left child, X_L and the right child, X_R. We now check whether $W(X_L) < off$. If it is, we move towards X_R with a new offset $off - W(X_L)$. Otherwise we move towards X_L and we keep the same offset as before. On the move towards the starting terminal we build a structure of a stack R, see section 3.1, used in the process of real time recovery. Thus the total time required is $O(|D|)$.

Theorem 3 *There exists a real-time $O(|D|)$ extra space algorithm, with $O(|D|)$ kick-off time, recovering consecutive symbols in grammar-based compressed files from any preprocessed position.*

4 Further discussion

In section 3 we have shown how to report in real-time consecutive symbols of a text, while dealing with its compressed version which requires only $O(|D|)$ extra space. This new algorithm can be used in time/space efficient processing of compressed files, e.g., in the context of text stream monitoring and pattern detection. In particular, a new real-time traversal of compressed texts can be nicely combined with a new small space (m^c, for any constant $c > 0$) real-time string matching algorithm presented in [6]. However, efficient traversal of compressed texts can be combined with almost any other variant of the string matching problem. For example, if we have an extra $O(|m|)$ space available we can decompress locally (without the need for decompressing the whole file) any fragment of the compressed text and then search it directly using standard efficient string matching algorithms. We also strongly believe that a new method can find its use in any application requiring streaming of textual data, e.g., a grammar-based compression method can be easily combined with error correcting codes. We hope that it will prove to be useful in real-time transmissions requiring high security.

References

[1] A. Amir, G. Benson, and M. Farach, Let sleeping files lie: Pattern matching in Z-compressed files, *Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1994, pp 705–714.

[2] A. Amir, G.M. Landau, and D. Sokol, Inplace Run-Length 2d Compressed Search, In Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'2000, San Francisco, pp. 817–818.

[3] M. Crochemore and W. Rytter, *Jewels of Stringology*, World Scientific, 2002.

[4] Laali Elkhalifa, InfoFilter: Complex Pattern Specification and Detection over Text Streams, *MSc Dissertation*, The University of Texas at Arlington, May 2004.

[5] P. Ferragina and G. Manzini, Opportunistic Data Structures with Applications. *Proc. 41st IEEE Symposium on Foundations of Computer Science*, (FOCS'00). Redondo Beach (CA), 2000, pp. 390–398.

[6] L. Gąsieniec and R. Kolpakov, Sublinear-space real-time string matching, In Proc. 15th Annual Symposium on Combinatorial Pattern Matching, CPM 2004, Istanbul, Turkey, July 5-7, 2004, LNCS 3109, pp. 117-129.

[7] L. Gąsieniec and I. Potapov. Time/Space Efficient Compressed Pattern Matching. In Proc. of 13th International Symposium on Fundamentals of Computation Theory, LNCS, Volume 2138, pp 138-152, Springer-Verlag, 2001.

[8] L. Gąsieniec and W. Rytter. Almost optimal fully compressed pattern matching. In Proceedings of *Data Compression Conference* (DCC'99), Snowbird, 1999, pp 316–325.

[9] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.

[10] J.C. Kieffer, A Survey of Advances in Hierarchical Data Compression, Technical Report, Department of Electrical & Computer Engineering, University of Minnesota, 2000.

[11] M. Miyazaki, A. Shinohara, and M. Takeda, An Improved Pattern Matching for Strings in Terms of Straight-Line Programs, *Journal of Discrete Algorithms*, Vol. 1(1), pp. 187–204, 2000.

[12] www.caplin.com, RTTP, a universal web protocol for streaming data, White Paper, February 2003.

[13] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, Speeding up pattern matching by text compression, In Proceedings of *4th Italian Conference on Algorithms and Complexity*, CIAC 2000, March 1-3, 2000, Rome, Italy, pp 306–315.

[14] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, pp. IT-23(3):337–343, 1977.