Algorithmica (2017) 77:336–348
DOI 10.1007/s00453-015-0068-9

Compressed Subsequence Matching and Packed Tree Coloring

Philip Bille[1] · **Patrick Hagge Cording**[1] · **Inge Li Gørtz**[1]

Received: 5 June 2014 / Accepted: 4 September 2015 / Published online: 30 September 2015

Abstract We present a new algorithm for subsequence matching in grammar compressed strings. Given a grammar of size n compressing a string of size N and a pattern string of size m over an alphabet of size σ, our algorithm uses $O(n + \frac{n\sigma}{w})$ space and $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot occ)$ or $O(n + \frac{n\sigma}{w} \log w + m \log N \cdot occ)$ time. Here w is the word size and occ is the number of minimal occurrences of the pattern. Our algorithm uses less space than previous algorithms and is also faster for $occ = o(\frac{n}{\log N})$ occurrences. The algorithm uses a new data structure that allows us to efficiently find the next occurrence of a given character after a given position in a compressed string. This data structure in turn is based on a new data structure for the tree color problem, where the node colors are packed in bit strings.

Keywords Straight line program · SLP · Compressed · Subsequence matching · Tree coloring · First colored ancestor

Philip Bille, Inge Li Gørtz: Supported in part by the The Danish Council for Independent Research and Natural Sciences Grant (DFF 1323-00178).

Philip Bille, Patrick Hagge Cording, Inge Li Gørtz: Supported in part by the Danish Research Council and the Danish Research Council under the Sapere Aude Program (DFF 4005-00267).

✉ Patrick Hagge Cording
phaco@dtu.dk

Philip Bille
phbi@dtu.dk

Inge Li Gørtz
inge@dtu.dk

[1] Technical University of Denmark, Kongens Lyngby, Denmark

1 Introduction

Subsequence matching is a variant of string pattern matching where an occurrence of the pattern in the text must contain all the characters of the pattern, but not necessarily contiguously. A pattern string P is a subsequence of another string S (the text) if P can be obtained by deleting characters in S. Just answering if P is a subsequence of S is easily solved in linear time by scanning S from left to right looking for the characters of P. In the *subsequence matching problem* the goal is to find and report the positions of all minimal substrings of S that contain P as a subsequence. A substring is said to be minimal if shortening it implies that P is no longer a subsequence of that substring. More formally, if P is a subsequence of the substring $S[i, j]$, then i, j is a minimal occurrence if P is not a subsequence of $S[i + 1, j]$ or $S[i, j - 1]$.

If the text is large and sufficiently repetetive, it might be useful to compress it. In this paper we consider the *compressed subsequence matching problem* where we are given a grammar $\mathcal{S}$ of size n compressing a string S of size N and a pattern string P of size m over an alphabet of size σ. We present a new algorithm for compressed subsequence matching which is space efficient and is faster than the previously fastest algorithm for a bounded number of occurrences. Our algorithm relies on a method that is different from the ones used by previous algorithms.

Subsequence matching is useful when searching sequential log data for a sequence of events that may be separated by other events. Say for instance that we are running a webserver and we want to know how often a visitor has found her way to subpage C through page A and then B. We then set $P = ABC$ and apply a subsequence matching algorithm to the contents of the log file. Many applications will automatically compress log data to save space, and so the bottleneck of the procedure becomes decompression of the data. In this case, processing the data without fully decompressing it, is crucial. Subsequence matching was also considered in relation to knowledge discovery and data mining [20].

Several algorithms have been presented for uncompressed strings [6,10,12,14,15, 20,27]. The fastest of these is due to Das et al. [15]. Since it is an online algorithm we may apply it to the compressed version without having to store the entire decompressed string, and we get an algorithm with running time $O(\frac{Nm}{\log m})$ that uses $O(n+m)$ space. The other algorithms are based on a forward–backward scanning approach. This also adopts to a $O(nm \cdot \text{occ})$-time and $O(n)$-space algorithm for SLPs. Our algorithm is also based on the scanning approach. Cégielski et al. [11] introduced the first algorithm designed for subsequence matching in a grammar-compressed string. Its runnning time is $O(nm^2 \log m + \text{occ})$ time and it uses $O(nm^2)$ space. Using a different approach, Tiskin improved the running time to $O(nm^{1.5} + \text{occ})$ [25] and later even further to $O(nm \log m + \text{occ})$ [26]. The space usage of his algorithms is $O(nm)$. The most recent improvement is due to Yamamoto et al. [28] who present an algorithm based on the ideas of Cegielski et al. that runs in $O(nm + \text{occ})$ time and $O(nm)$ space. All results are summarized in Table 1.

Our algorithm assumes that the input grammar is a Straight Line Program (SLP). An SLP is an acyclic grammar in Chomsky normal form, i.e., each nonterminal production rule expands to two other rules and generates one string only. Any grammar producing a single string can be transformed to an SLP with linear overhead so our results hold for

Table 1 Time and space complexities of algorithms for compressed subsequence matching

Time complexity	Space complexity	Author(s)
$O(\frac{Nm}{\log m})$	$O(n+m)$	Das et al. [15]
$O(nm \cdot \text{occ})$	$O(n)$	Folklore
$O(nm^2 \log m + \text{occ})$	$O(nm^2)$	Cégielski et al. [11]
$O(nm^{1.5} + \text{occ})$	$O(nm)$	Tiskin [25]
$O(nm \log m + \text{occ})$	$O(nm)$	Tiskin [26]
$O(nm + \text{occ})$	$O(nm)$	Yamamoto et al. [28]
$O(n + \frac{n\sigma}{w} + m \log N \log w \cdot \text{occ})$	$O(n + \frac{n\sigma}{w})$	This paper
$O(n + \frac{n\sigma}{w} \log w + m \log N \cdot \text{occ})$		

For all complexities occ ≥ 1

grammar-compressed strings in general. Moreover, SLPs are widely studied because they model many well-known compression schemes, such as LZ77 [29], LZ78 [30], and Re-Pair [19] with little overhead [13,22]. The following theorem is the main result of this work.

Theorem 1 *Given an SLP $\mathcal{S}$ of size n compressing a string S of size N and a pattern P of size m over an alphabet of size σ, compressed subsequence matching can be solved in $O(n + \frac{n\sigma}{w})$ words of space and time*

(i) $O(n + \frac{n\sigma}{w} + m \log N \log w \cdot occ)$*, or*
(ii) $O(n + \frac{n\sigma}{w} \log w + m \log N \cdot occ)$

in the word RAM model with word size $w \geq \log N$, and where occ is the number of minimal occurrences of P in S.

Our new algorithm uses less space (linear in n if $\sigma \leq w$) and is also faster than the previously fastest algorithm for few occurrences when $\sigma \leq m$. Particularly, solution (ii) is faster if the number of occurrences is bounded by $o(\frac{n}{\log N})$. Note that we can, in $O(n+m)$ expected time and using $O(m)$ additional extra space, guarantee that $\sigma \leq m$ always holds with hashing.

The algorithm is based on the idea of a simple algorithm for subsequence matching in uncompressed strings which basically scans the string for occurrences of the pattern. We speed up the scanning on compressed strings by introducing the first data structure for SLPs that supports labelled successor queries. The answer to a labelled succesor query $\text{LS}(i, c)$ on a string is the index of the first character c occurring after position i in the string. An essential part of this data structure is a new data structure for the tree color problem. This problem is to preprocess a tree where each node is colored by zero or more colors, such that given a node v and a color c, we may efficiently answer a first colored ancestor query, i.e., compute the lowest ancestor of v with color c. Additionally, this data structure also supports a new type of query we call the last colored ancestor. Here the query is two nodes u and v and a color c, and the answer is the highest node on the path from u to v with color c. These results may be of independent interest.

This paper is organized such that we start by describing our new result for the tree color problem (after a section of preliminaries), then move on to the labelled successor data structure, and finally describe the algorithm for subsequence matching.

2 Preliminaries

2.1 Bit Strings

We will use bit strings to represent sets. In a bit string $B = b_1 b_2 \dots b_u$ representing a set $\mathcal{B}$ of elements from a universe of size u, $b_i = \mathtt{1}$ iff element i is in $\mathcal{B}$. $B = [\mathtt{0}]^u$ denotes the empty set. The operators $\wedge$, $\vee$, and $\oplus$ denote the bitwise AND, OR, and exclusive OR (XOR) of two bit strings. The negation of a bit string B is $\overline{B}$. A *summary* B_s of k bit strings $B_1, B_2, \dots, B_k$ of equal length is $B_s = B_1 \vee B_2 \vee \dots \vee B_k$. For a bit string of length w we assume that the mask of any constant can be computed in $O(1)$ time. Given a bit string $B = b_1 b_2 \dots b_w$, b_1 is the most significant bit. The index of the least significant set bit can be found in $O(1)$ time from $\log_2((\overline{B}+1) \wedge B)$. Finding the most significant set bit is more elaborate, but can also be done $O(1)$ time [18]. An $n \times m$ bit matrix may be transposed in $O(w \log w)$ time if $n \leq w$ and $m \leq w$ [24].

2.2 Trees

In this paper all trees are rooted, ordered, and have labels on the nodes. The number of nodes in a tree T is t. We denote by $T(v)$ the subtree rooted at v containing all descendants of v. The size $|T(v)|$ is the number of nodes in the subtree $T(v)$ including v. If u is a node in the subtree $T(v)$ we write $u \in T(v)$. If T is a binary tree we denote the left and right child of a node v by $left(v)$ and $right(v)$.

A heavy path decomposition [23] decomposes T into disjoint paths. Nodes are classified as either heavy or light and the decomposition is defined as follows. The root is light. For each internal node v, its heavy child w is the node for which $T(w)$ is of maximum size among the subtrees rooted at children of v. The other children of v are light. Edges are also classified as heavy or light. An edge going into a heavy node is heavy and likewise for light nodes. The heavy path decomposition ensures the property that $\frac{1}{2}|T(v)| > |T(u)|$ for any light child u of v. This means that there are $O(\log t)$ light edges on any path from the root to a leaf. The heavy path decomposition can be computed in $O(t)$ time and space.

Given a binary tree T rooted at a node r, $t > 1$, and a parameter $1 \leq x \leq t$, we may partition T into at most t/x clusters such that for a fixed constant c, the size of any cluster is at most cx [3,5] (see also [1] for a full proof). Two clusters overlap in at most one node, and a node is called a boundary node if it is part of more than one cluster. Any cluster has at most two boundary nodes, and a boundary node is either a leaf or the root in the subtree that is the cluster. The sum of nodes in all clusters is $O(t)$. The tree obtained by repeatedly contracting edges between two nodes if one of them is not a boundary node is called the macro tree. In other words, the macro tree is the tree consisting only of boundary nodes. A cluster partition can be found in $O(t)$ time.

$$\begin{aligned}X_1 &= \mathtt{a}\\ X_2 &= \mathtt{b}\\ X_3 &= X_1X_2\\ X_4 &= X_3X_1\\ X_5 &= X_4X_3\\ X_6 &= X_5X_4\\ X_7 &= X_6X_5\end{aligned}$$

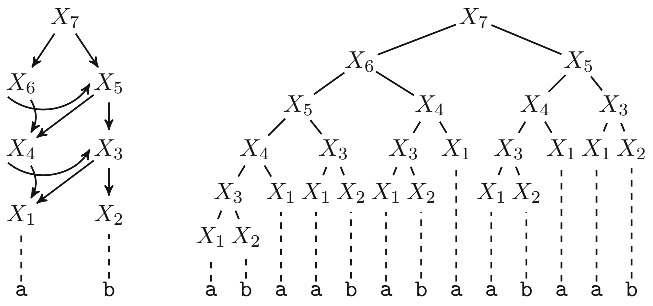

Fig. 1 An SLP compressing the string abaababaabaab, its corresponding DAG, and its derivation tree

The answer to a level ancestor query LA(v, d) on T is the ancestor of v with depth d. A linear space data structure that answers an LA query in $O(1)$ time can be computed for T in $O(t)$ time [16] (see also [2,7,8]).

2.3 Straight Line Programs

A Straight Line Program $\mathcal{S}$ is a context-free grammar in Chomsky normal form with n production rules that produce a single string S of length N. We represent the SLP as a rooted, ordered, and node-labelled directed acyclic graph (DAG) with outdegree 2 and we will refer to production rules as nodes in the DAG. A depth-first left-to-right traversal starting from a node v in the DAG produces the string $S(v)$ of length $|S(v)|$. The tree that emerges from the traversal we call the derivation tree. An example is shown in Fig. 1. We denote the left and right children of v for $left(v)$ and $right(v)$, respectively. Furthermore, the height of the SLP is the length of the longest path going from the root to a terminal node and is denoted by h.

We may access a character $S[i]$ in $O(h)$ time by storing $|S(v)|$ for each node v in the SLP, and simulate a top–down search of the derivation tree. Doing so yields a unique path from the root of $\mathcal{S}$ to the terminal node labelled $S[i]$. There is also a linear space data structure that supports random access in SLPs in $O(\log N)$ time [9]. A key technique used in this data structure is the extension of the heavy path decomposition of trees to SLPs which we will also use in our data structure. For each node $v \in \mathcal{S}$, we select the child of v that derives the longest string to be a heavy node. The other child is light. Heavy and light edges are defined as in the decomposition of trees. Whereas applying this technique to a tree results in a decomposition into disjoint paths, it will result in a decomposition into disjoint trees when applied to an SLP (see Fig. 2). We denote this set of trees by the heavy forest $\mathcal{H}$ of the SLP. This decomposition ensures that the number of light edges on any path from the root to a terminal node is $O(\log N)$. Hence, on any path from the root of the SLP to a terminal node, we visit at most $\log N$ trees from $\mathcal{H}$. When accessing a character using the data structure of [9] we may also report the entry and exit nodes for each tree visited on the unique root-to-terminal path that emerges from the query.

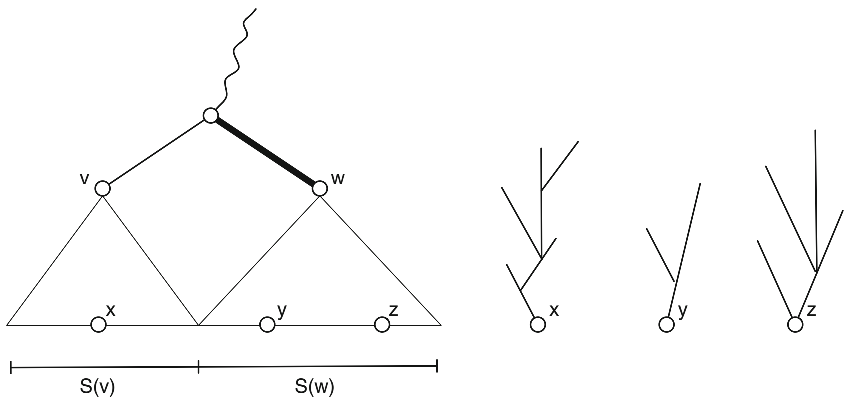

Fig. 2 An example of the heavy path decomposition of SLPs. (*Left*) a node in $\mathcal{S}$ (here depicted as a tree) with children v and w where the edge to w is selected as heavy because $S(w) \geq S(v)$. The nodes x, y, and z are terminals in $\mathcal{S}$. (*Right*) the heavy forest obtained by removing light edges from $\mathcal{S}$. The trees are rooted in terminals of $\mathcal{S}$ and can therefore be seen as growing upwards

3 Packed Tree Color Problems

In a colored tree, each node is colored by zero or more colors from the set $\{1, \ldots, \sigma\}$. A packed colored tree is a colored tree where the colors of each node v are given as a bit string $C(v)$ where $C(v)[c] = 1$ iff v is colored c. In this section we consider the *packed tree color problem* which is to preprocess a packed colored tree T to support first and last colored ancestor queries. The answer to a first colored ancestor query FIRSTCOLOR(v, c) is the lowest ancestor of v with color c, and the answer to a last colored ancestor query LASTCOLOR(u, v, c) is the highest node with color c on the path from u to v, where we always assume that u is an ancestor of v. Throughout this section we will use the following notation to distinguish results. If a data structure requires $p(t)$ time to build, uses $s(t)$ space, and supports FIRSTCOLOR and LASTCOLOR queries in $q(t)$ time, then the the triple $\langle p(t), s(t), q(t)\rangle$ refers to the solution.

Solutions to the tree color problem for trees that are not packed may be applied to packed trees. All known solutions focus entirely on supporting FIRSTCOLOR queries [4,16,17,21]. A simple solution that supports FIRSTCOLOR queries in $O(1)$ time is to store the answer for every color in every node. This yields a $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ solution. The currently best known trade-off for the tree color problem is $\langle O(t + D), O(t + D), O(\log w)\rangle$ [21], where $D = \sum_{v \in T} \sum_{i=1}^{\sigma} C(v)[i]$ is the accumulated number of colors used.

Our motivation for revisiting this problem is twofold. First we have that $D = O(t\sigma)$ in our application and we are striving for a space bound that is in $o(t\sigma)$. Second we want to support LASTCOLOR queries.

In this section we present three solutions to the packed tree coloring problem and combine them to a data structure with a new and desireable time-space trade-off.

3.1 A $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ Solution

We store the result of a FIRSTCOLOR(v, c) query for every node and color. For each color, let the induced c-colored subtree be the tree obtained by deleting all nodes that are not colored by color c except the root. Build a levelled ancestor data structure for each induced colored subtree.

The result of a FIRSTCOLOR query is precomputed. A LASTCOLOR(u, v, c) query is answered as follows. If FIRSTCOLOR$(v, c) =$ FIRSTCOLOR(u, c) then there is not a node with color c on the path from u to v. If FIRSTCOLOR$(v, c) \neq$ FIRSTCOLOR(u, c) then let v' and u' be the nodes corresponding to FIRSTCOLOR(v, c) and FIRSTCOLOR(u, c) in the induced c-colored subtree. The answer to LASTCOLOR(u, v, c) is then the answer to LA$(v', depth(u') + 1)$ in the induced c-colored subtree.

The results of FIRSTCOLOR queries can be found and stored using $O(t\sigma)$ time and space. The induced colored subtrees can be computed in $O(t\sigma)$ time and use $O(D) = O(t\sigma)$ space. A FIRSTCOLOR query clearly takes $O(1)$ time. For a LASTCOLOR query, we perform two FIRSTCOLOR queries and one LA query, each of which takes constant time.

Lemma 1 *The packed tree color problem can be solved using $O(t\sigma)$ preprocessing time and space, and $O(1)$ query time.*

3.2 A $\langle O(t + \frac{t\sigma}{w}), O(t + \frac{t\sigma}{w}), O(\log t)\rangle$ Solution

We fix a heavy path decomposition of T. For each heavy path $p = v_1, v_2, \ldots, v_k$, where v_1 is the highest node on the path, we build a balanced binary tree T_p having the nodes of p as leaves ordered from left to right. For each node v in T_p we store a summary $B(v)$ of the colors of its children. For each heavy path p we also store a summary $P(v_i)$ of colors on the path prefix $v_1 \ldots v_i$ for every v_i on p.

For answering a FIRSTCOLOR(v, c) query, let $p = v_1, v_2, \ldots, v_k$ be the heavy path containing v and let $v_i = v$ for some $1 \leq i \leq k$. If $P(v_i)[c] = \mathtt{1}$ we find the lowest ancestor x of v_i in T_p for which $B(left(x))[c] = \mathtt{1}$ and $v_i \notin T_p(left(x))$. The answer to the query is then the rightmost leaf in $T_p(left(x))$ with color c. If $P(v_i)[c] = \mathtt{0}$ we repeat the procedure with $v_i = parent(v_1)$, i.e., we jump to the previous heavy path, until we find the first colored ancestor or we reach the root of T.

A LASTCOLOR(u, v, c) query is handled in a similar way. We first find the highest light node w on the path from u to v for which $P(parent(w))[c] = \mathtt{1}$. Let p be the heavy path containing $parent(w)$. Now there are three cases. If u is not on p, the answer to the query is the leftmost leaf in T_p that has color c. If p contains u, the answer is the leftmost leaf with color c to the right of u in T_p, if such a node exists. If it does not exist, we repeat the first step for the second highest light node w' between u and v for which $P(parent(w'))[c] = \mathtt{1}$.

The heavy path decomposition of T can be found and stored in $O(t)$ time and space. Since the paths of the heavy path decomposition are disjoint, the total number of leaves in the binary summary trees is t, so the total number of nodes in the trees is

$O(t)$. We store $O(t)$ summary bit vectors of size $O(\frac{\sigma}{w})$ using a total of $O(\frac{t\sigma}{w})$ space. We use $O(\frac{t\sigma}{w})$ bitwise OR operations to create the summaries in a bottom up fashion. In total, preprocessing time and space usage is $O(t + \frac{t\sigma}{w})$.

For both queries we visit at most $\log t$ heavy paths. When the path with the answer has been found we walk up the binary tree and then down again. Since the tree is balanced and has at most t leaves, this takes $O(\log t)$ time. For LASTCOLOR queries we do this at most twice. The query time for FIRSTCOLOR and LASTCOLOR queries is therefore $O(\log t)$ time.

Lemma 2 *The packed tree color problem can be solved using* $O(t+\frac{t\sigma}{w})$ *preprocessing time and space, and* $O(\log t)$ *query time.*

3.3 A $\langle O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w}), O(t + \frac{t\sigma}{w} + \frac{t^2}{w}), O(\frac{t}{w})\rangle$ Solution

Let $v_1, \ldots, v_t$ be the nodes of T in pre-order. We will represent T as a $\sigma \times t$ bit matrix M. Let c be a color from the set of colors $\{1, \ldots, \sigma\}$. In row c of M we store a bit string where bit i is $\mathtt{1}$ iff v_i has color c. For each node v_i we also store a bit string $A(v_i)$ where bit j is $\mathtt{1}$ iff v_j is an ancestor of v_i.

We construct this data structure from a packed colored tree as follows. Assume that the bit strings representing the node colorings form a $t \times \sigma$ matrix where row i is the colorings of node v_i. We transpose this matrix to get M. To do this we partition the matrix into a $\frac{t}{w} \times \frac{\sigma}{w}$ matrix (assume w.l.o.g. that w divides t and σ), transpose each $w \times w$ submatrix as described in [24], and transpose the $\frac{t}{w} \times \frac{\sigma}{w}$ matrix to get M. To compute the ancestor bit strings first set $A(root(T)) = [\mathtt{0}]^t$. For all other nodes v_i, where v_j is the parent of v_i, set $A(v_i) = A(v_j) \vee 2^j$.

We answer a FIRSTCOLOR(v, c) as follows. Let $R = M[c] \wedge A(v)$. Now R is a bit string representing the set of ancestors of v with color c. Since the nodes have pre-order indices, the answer to the query is v_i, where i is the index of the least significant set bit in R.

To answer a LASTCOLOR(v, u, c) query we start by computing R the same way as above. We then set the first $i - 1$ bits of R to $\mathtt{0}$, where i is the index of u. The answer to the query is the most significant set bit of R.

The $\sigma \times t$ bit matrix M can be packed in words and therefore uses $O(\frac{t\sigma}{w})$ space. Similarly, the ancestor bit strings use $O(\frac{t^2}{w})$ space. Transposing a $w \times w$ matrix takes $O(w \log w)$ time, and since there are $\frac{t\sigma}{w^2}$ submatrices of this size in the color bit matrix, the total time spent for all submatrices is $O(\frac{t\sigma \log w}{w})$. Transposing the $\frac{t}{w} \times \frac{\sigma}{w}$ matrix takes $O(\frac{t\sigma}{w})$ time. Computing the ancestor bit strings takes $O(\frac{t^2}{w})$ time.

The size of R is $O(\frac{t}{w})$, so finding the first non-zero word takes $O(\frac{t}{w})$ time. Determining the least or most significant set bit of a word is done in $O(1)$ time. Thus, the query time for both a FIRSTCOLOR and a LASTCOLOR query is $O(\frac{t}{w})$.

Lemma 3 *The packed tree color problem can be solved using* $O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w})$ *preprocessing time,* $O(t + \frac{t\sigma}{w} + \frac{t^2}{w})$ *space, and* $O(\frac{t}{w})$ *query time.*

3.4 Combining the Solutions

We now show how to combine the previously described solutions to get $\langle O(t + \frac{n\sigma}{w}), O(t + \frac{n\sigma}{w}), O(\log w)\rangle$ and $\langle O(t + \frac{t\sigma \log w}{w}), O(t + \frac{t\sigma}{w}), O(1)\rangle$ trade-offs. This is achieved by doing a cluster partioning of the tree.

First we convert T to a binary tree T'. Then we partition T' into $O(\frac{t}{w})$ clusters, i.e., each cluster has size $O(w)$. For each cluster C, where one boundary node is a leaf in the cluster and the other is the root of the cluster, we make a summary of the colors of the nodes on the path from the root to the leaf. The summary is stored in the macro tree node that corresponds to the leaf boundary node of C. Apply the $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ solution to the macro tree, and apply either the $\langle O(t + \frac{t\sigma}{w}), O(t + \frac{t\sigma}{w}), O(\log t)\rangle$ solution or the $\langle O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w}), O(t + \frac{t\sigma}{w} + \frac{t^2}{w}), O(\frac{t}{w})\rangle$ solution to each cluster using the original colors.

Here is how we answer a FIRSTCOLOR(v, c) query. Let C_v be the cluster containing v. First we ask for FIRSTCOLOR(v, c) in C_v. If the answer is a node in C_v, we are done. If it is undefined, we find the node r in the macro tree corresponding to the root of C_v. We check if r has color c in the macro tree and otherwise ask for $w =$ FIRSTCOLOR(r, c) in the macro tree. In the cluster C_w having w as a leaf boundary node we then check if w has color c and otherwise ask for FIRSTCOLOR(w, c) in C_w.

We answer a LASTCOLOR(u, v, c) query as follows. Assume that $u \neq v$ and let C_u and C_v be the clusters containing u and v. If $C_u = C_v$ then the answer is LASTCOLOR(u, v, c) in the cluster containing u and v. If $C_u \neq C_v$, let w be the leaf boundary node of C_u where $v \in T(w)$. We now proceed in three steps. First, we ask for LASTCOLOR(u, w, c) in C_u. If the query returns a node, this is also the answer to the LASTCOLOR(u, v, c) query. If the answer in the first step is undefined we ask for $z =$ LASTCOLOR$(w, root(C_v), c)$ in the macro tree to locate the highest cluster with a node with color c between u and v. The answer to the query is then LASTCOLOR$(root(C_z), z, c)$ on C_z. If the first two steps fail, the answer to a query is LASTCOLOR$(root(C_v), v, c)$.

The cluster partition can be computed in linear time. To compute the cluster path summaries we OR the color bit strings of the nodes on the path, hence we spend $O(\lceil \frac{\sigma}{w} \rceil)$ time per node. On a cluster of size w we thus spend $O(\max\{w, \sigma\})$ time, totalling $O(\max\{t, \frac{t\sigma}{w}\})$ time for the $O(\frac{t}{w})$ clusters. Since the macro tree has $O(\frac{t}{w})$ nodes the preprocessing time and space to apply the $\langle O(t\sigma), O(t\sigma), O(1)\rangle$ solution becomes $O(\frac{t\sigma}{w})$. To answer a query we perform a constant number of FIRSTCOLOR and LASTCOLOR queries on the macro tree and clusters. Therefore the total time to perform queries on the macro tree is $O(1)$ time. To get (i) we apply the $\langle O(t + \frac{t\sigma}{w}), O(t + \frac{t\sigma}{w}), O(\log t)\rangle$ solution to clusters. Since a cluster has size $O(w)$ we use a total of $O(\log w)$ time performing queries on clusters. To get (ii) we apply the $\langle O(t + \frac{t\sigma \log w}{w} + \frac{t^2}{w}), O(t + \frac{t\sigma}{w} + \frac{t^2}{w}), O(\frac{t}{w})\rangle$ solution to clusters. Again, since clusters have size $O(w)$ we use a total of $O(1)$ time performing queries on clusters. Preprocessing time and space for the cluster data structures follow because the sum of the sizes of clusters is $O(t)$.

Theorem 2 *The packed tree color problem can be solved using $O(t + \frac{t\sigma}{w})$ space,*

(i) $O(t + \frac{t\sigma}{w})$ *preprocessing time, and* $O(\log w)$ *query time, or*
(ii) $O(t + \frac{t\sigma}{w} \log w)$ *preprocessing time, and* $O(1)$ *query time.*

4 Labelled Successor Data Structure for SLPs

The answer to a labelled successor $\textsc{ls}(i, c)$ query on a string S is the index of the first occurrence of the character c after position i in S. More formally, the answer to $\textsc{ls}(i, c)$ is an index j such that $S[j] = c$, $j > i$, and $S[k] \neq c$ for $k = i+1, \ldots, j-1$.

In this section we present a data structure that supports $\textsc{ls}(i, c)$ queries on an SLP. This is the first data structure dedicated to solving this problem on SLPs. Alternatively, we may build the random access data structure of [9] and then answer an $\textsc{ls}(i, c)$ query by doing a random access query for position i followed by a linear scan to find the first occurrence of c. This yields a query time of $O(\log N + j - i)$ while using $O(n)$ space for the data structure.

Our data structure combines the random access data structure of [9] with a new way of navigating the SLP based on the characters of substrings. For the latter we will utilize our result for the packed tree color problem described in the previous section.

The basic idea is to store a bit string for each node $v \in \mathcal{S}$ that summarizes which characters are generated by $S(v)$. We first seach for position i in S and let p be the unique path in $\mathcal{S}$ defining $S[i]$. We then walk up p until reaching a node u where $right(u)$ generates a string that contains c and $right(u)$ is not on p. Then we walk down from $right(u)$ using the summaries to locate the leftmost terminal descending from $right(u)$ that generates c. This algorithm requires $O(n + \frac{n\sigma}{w})$ space and $O(h)$ time to find $\textsc{ls}(i, c)$.

To speed things up we fix a heavy path decomposition of the SLP to get a heavy forest and build the random access data structure of [9]. Now p is a sequence of entry and exit points in the trees of the heavy forest. When we walk up p we enter a tree in an exit node and have to walk away from the root to the first node whose right child generates a string that contains c before reaching the entry node. This is equivalent to a LASTCOLOR query. When we walk down to find $\textsc{ls}(i, c)$ we enter a tree and have to walk towards the root to find either the first ancestor whose left child generates a string that contains c or the highest ancestor whose right child generates c. This is equivalent to a FIRSTCOLOR and a LASTCOLOR query, respectively.

In the remainder of this section we give the details of the data structure. An example of the data structure and a query is given in Fig. 3.

Theorem 3 *There is a data structure supporting labelled successor (and predecessor) queries on a string of size N over an alphabet of size* σ *compressed by an SLP of size n in the word RAM model with word size* $w \geq \log N$ *using* $O(n + \frac{n\sigma}{w})$ *space and*

(i) $O(n + \frac{n\sigma}{w})$ *preprocessing time, and* $O(\log N \log w)$ *query time, or*
(ii) $O(n + \frac{n\sigma}{w} \log w)$ *preprocessing time, and* $O(\log N)$ *query time.*

Proof We first apply the construction of [9], and let $\mathcal{H}$ be the heavy forest obtained from the heavy path decomposition of $\mathcal{S}$. For each node v in $\mathcal{S}$ with children $left(v)$ and $right(v)$ we store two bit strings $L(v)$ and $R(v)$ summarizing the characters in $S(left(v))$ and $S(right(v))$. Specifically, $L(v)$ summarizes the characters in

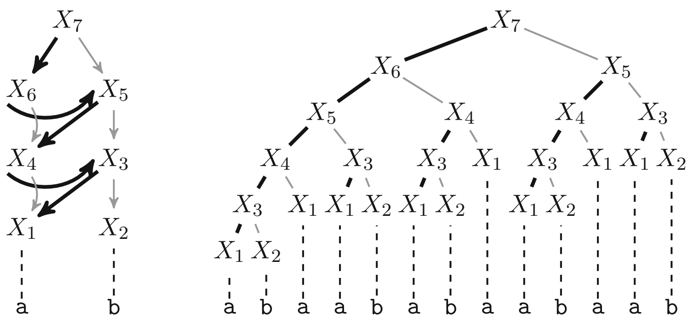

Fig. 3 An example of the data structure for the SLP shown in Fig. 1. The heavy edges are marked in both the SLP and its parse tree. The resulting heavy forest contains the path from X_1 to X_7 (X_1 is the root) and X_2 (a tree with one node). For the non-terminals we have that $L(X_i) = \texttt{00}$ for $i = 3 \ldots 7$, and $R(X_3) = \texttt{01}$, $R(X_4) = \texttt{10}$, $R(X_5) = \texttt{11}$, $R(X_6) = \texttt{11}$, $R(X_7) = \texttt{11}$. Suppose we want to answer the query $\text{LS}(2, \texttt{b})$. After a random access query for position 2 the entry and exit points of the heavy trees are (X_7, X_3), (X_2, X_2). The algorithm then performs the queries $\text{LASTCOLOR}(X_2, X_2, \texttt{b})$ and $\text{LASTCOLOR}(X_3, X_7, \texttt{b})$. The result of the latter query is X_5 because X_5 is the highest node on the path from X_7 to X_3 in the heavy forest that has the color b. This means that the subgraph rooted in $right(X_5)$ generates a b. FIRSTCOLOR queries are then performed to identify the occurrence of the first b in $\mathcal{S}(right(X_5))$

$S(left(v))$ if the edge from v to $left(v)$ is light. If the edge is heavy then $L(v) = [\texttt{0}]^\sigma$. $R(v)$ is defined analogously. These summaries are used for determining where to exit a heavy tree when searching for some character. When following a heavy edge, say from v to $left(v)$, we do not exit a tree, so therefore $L(v)$ is set to $[\texttt{0}]^\sigma$. For each tree in $\mathcal{H}$ we build two data structures for the packed tree color problem. One where the L bit strings serve as colors and one where the R bit strings serve as colors.

We answer an $\text{LS}(i, c)$ query as follows. First we access the character $S[i]$ using the random access data structure. We now have the entry and exit points of the heavy trees in $\mathcal{H}$ on the unique path p describing $S[i]$. Let $T_1, \ldots, T_k \in \mathcal{H}$ be a sequence of trees on p in the order they are visited when starting from the root and ending in the terminal generating $S[i]$, and let $(v_1, u_1), \ldots, (v_k, u_k)$ be the entry and exit nodes for each tree in the sequence. Using the packed tree color data structure for the R colors, we repeat $\text{LASTCOLOR}(u_l, v_l, c)$ for $l = k$ down to some j until $\text{LASTCOLOR}(u_j, v_j, c)$ is not undefined. Let $w = right(\text{LASTCOLOR}(u_j, v_j, c))$. We now search for the first occurrence of c in $S(w)$. Let T_s be the tree in $\mathcal{H}$ that contains the node w, then the search proceeds in three steps. First, we ask for $v = \text{FIRSTCOLOR}(w, c)$ in T_s in the data structure for L colors and restart the search from $left(v)$. If the query $\text{FIRSTCOLOR}(w, c)$ is undefined we continue to the next step. In the second step we check if $root(T_s)$ generates c. If it does, we now have a unique set of entry and exit nodes in the trees of $\mathcal{H}$ that constitutes a path to a terminal that generates the first c after position i. The answer to the $\text{LS}(i, c)$ query is the index of this c which we retrieve using the random access data structure. Finally, if $root(T_s)$ does not generate c we ask for $v = \text{LASTCOLOR}(w, root(T_s), c)$ in T_s in the data structure for R colors, and restart the search from $right(v)$.

The data structure uses $O(n + \frac{n\sigma}{w})$ space because the random access data structure uses linear space and the bit strings L and R use $O(\frac{n\sigma}{w})$ space. The random access data structure, including the heavy path decomposition, takes $O(n)$ time to compute and the L and R values are computed using $O(\frac{n\sigma}{w})$ OR operations in a bottom up fashion. Therefore, this part of the data structure is computed in $O(n + \frac{n\sigma}{w})$ time.

To get Theorem 3(i) we use the packed tree color data structure of Theorem 2(i) for the trees in $\mathcal{H}$ and likewise for (ii). Since the trees are disjoint, the preprocessing time and space becomes as in the Theorem 3.

For the query, we first do one random access query that takes $O(\log N)$ time, then we perform at most $\log N$ LASTCOLOR queries walking up the SLP and at most $2 \log N$ FIRSTCOLOR and LASTCOLOR queries locating the labelled successor. Finally, retrieving the index also takes $O(\log N)$ time using the random access data structure. □

5 Subsequence Matching

We will now use the labelled successor data structure to obtain a subsequence matching algorithm for SLPs. Our algorithm is based on the folklore algorithm for subsequence matching which works as follows (see also [15,20]). First we find the minimal prefix $S[1 \dots j]$ that contains P as a subsequence. This is done by reading S left to right while searching for the characters of P one at a time. We then find the minimal suffix $S[i \dots j]$ of the prefix $S[1 \dots j]$ that contains P. Similarly, this is done by scanning the prefix right to left. Now $S[i \dots j]$ is the first minimal occurrence of P. To find the next minimal occurrence we repeat this process for the suffix $S[i+1 \dots N]$. It can be shown that this algorithm finds all minimal occurrences of P in $O(Nm)$ time.

By using our labelled successor data structure described in the previous section we speed up the procedure of finding some specific character of P. Assume we have matched $P[1 \dots k]$ to $S[1 \dots j]$ such that $P[k] = S[j]$. Instead of doing a linear scan of $S[j+1 \dots N]$ to find $P[k+1]$ we ask for the next occurrence of $P[k+1]$ using LS$(j, P[k+1])$.

For each occurrence of P we perform $O(m)$ labelled successor (and labelled predecessor) queries, and we also have to construct the data structures to support these. By applying the results of Theorem 3 we get Theorem 1.

References

1. Abiteboul, S., Alstrup, S., Kaplan, H., Milo, T., Rauhe, T.: Compact labeling scheme for ancestor queries. SIAM J. Comput. **35**(6), 1295–1309 (2006)
2. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Montanari, U., Rolim, J.P., Welzl, E. (eds.) Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 1853, pp. 73–84. Springer, Berlin, Heidelberg (2000)
3. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Minimizing diameters of dynamic trees. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) Automata, Languages and Programming. Lecture Notes in Computer Science, vol. 1256, pp. 270–280. Springer, Berlin, Heidelberg (1997)
4. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, pp. 534–543. IEEE Computer Society, Washington, DC (1998)
5. Alstrup, S., Secher, J.P., Spork, M.: Optimal on-line decremental connectivity in trees. Inf. Process. Lett. **64**(4), 161–164 (1997)
6. Baeza-Yates, R.A.: Searching subsequences. Theor. Comput. Sci. **78**(2), 363–376 (1991)
7. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. **321**(1), 5–12 (2004)

8. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. **48**(2), 214–230 (1994)
9. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 373–389. SIAM, San Francisco (2011)
10. Boasson, L., Cegielski, P., Guessarian, I., Matiyasevich, Y.: Window-accumulated subsequence matching problem is linear. Ann. Pure Appl. Log. **113**(1), 59–80 (2001)
11. Cégielski, P., Guessarian, I., Lifshits, Y., Matiyasevich, Y.: Window subsequence problems for compressed texts. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) Computer Science-Theory and Applications. Lecture Notes in Computer Science, vol. 3967, pp. 127–136. Springer, Berlin, Heidelberg (2006)
12. Cégielski, P., Guessarian, I., Matiyasevich, Y.: Multiple serial episodes matching. Inf. Proc. Lett. **98**(6), 211–218 (2006)
13. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Trans. Inf. Theory **51**(7), 2554–2576 (2005)
14. Crochemore, M., Melichar, B., Troníček, Z.: Directed acyclic subsequence graph—overview. J. Discret. Algorithms **1**(3), 255–280 (2003)
15. Das, G., Fleischer, R., Gasieniec, L., Gunopulos, D., Kärkkäinen, J.: Episode matching. In: Apostolico, A., Hein, J. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 1264, pp. 12–27. Springer, Berlin, Heidelberg (1997)
16. Dietz, P.F.: Finding level-ancestors in dynamic trees. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) Algorithms and Data Structures. Lecture Notes in Computer Science, vol. 519, pp. 32–40. Springer, Berlin, Heidelberg (1991)
17. Ferragina, P., Muthukrishnan, S.: Efficient dynamic method-lookup for object oriented languages. In: Diaz, J., Serna, M. (eds.) European Symposium on Algorithms. Lecture Notes in Computer Science, vol. 1136, pp. 107–120. Springer, Berlin, Heidelberg (1996)
18. Fredman, M.L., Willard, D.E.: Surpassing the information theoretic bound with fusion trees. J. Comput. Syst. Sci. **47**(3), 424–436 (1993)
19. Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. P. IEEE. **88**(11), 1722–1732 (2000)
20. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovery of frequent episodes in event sequences. Data Min. Knowl. Disc. **1**(3), 259–289 (1997)
21. Muthukrishnan, S., Müller, M.: Time and space efficient method-lookup for object-oriented programs. In: Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 42–51. Society for Industrial and Applied Mathematics, Atlanta (1996)
22. Rytter, W.: Application of Lempel–Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. **302**(1), 211–222 (2003)
23. Sleator, D.D., Endre Tarjan, R.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3), 362–391 (1983)
24. Thorup, M.: Randomized sorting in $O(n \log \log n)$ time and linear space using addition, shift, and bit-wise boolean operations. J. Algorithms **42**(2), 205–230 (2002)
25. Tiskin, A.: Faster subsequence recognition in compressed strings. J. Math. Sci. **158**(5), 759–769 (2009)
26. Tiskin, A.: Towards approximate matching in compressed strings: local subsequence recognition. In: Kulikov, A., Vereshchagin, N. (eds.) Computer Science-Theory and Applications, vol. 6651, pp. 401–414. Springer, Berlin, Heidelberg (2011)
27. Troníček, Z.: Episode matching. In: Amir, A. (ed.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 2089, pp. 143–146. Springer, Berlin, Heidelberg (2001)
28. Yamamoto, T., Bannai, H., Inenaga, S., Takeda, M.: Faster subsequence and dont-care pattern matching on compressed texts. In: Giancarlo, R., Manzini, G. (eds.) Combinatorial Pattern Matching. Lecture Notes in Computer Science, vol. 6661, pp. 309–322. Springer, Berlin, Heidelberg (2011)
29. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
30. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)