RANDOM ACCESS TO GRAMMAR-COMPRESSED STRINGS AND TREES[*]

PHILIP BILLE[†], GAD M. LANDAU[‡], RAJEEV RAMAN[§], KUNIHIKO SADAKANE[¶], SRINIVASA RAO SATTI[‖], AND OREN WEIMANN[‡]

Abstract. Grammar-based compression, where one replaces a long string by a small context-free grammar that generates the string, is a simple and powerful paradigm that captures (sometimes with slight reduction in efficiency) many of the popular compression schemes, including the Lempel–Ziv family, run-length encoding, byte-pair encoding, Sequitur, and Re-Pair. In this paper, we present a novel grammar representation that allows efficient random access to any character or substring without decompressing the string. Let S be a string of length N compressed into a context-free grammar $\mathcal{S}$ of size n. We present two representations of $\mathcal{S}$ achieving $O(\log N)$ random access time, and either $O(n \cdot \alpha_k(n))$ construction time and space on the pointer machine model, or $O(n)$ construction time and space on the RAM. Here, $\alpha_k(n)$ is the inverse of the kth row of Ackermann's function. Our representations also efficiently support decompression of any substring in S: we can decompress any substring of length m in the same complexity as a single random access query and additional $O(m)$ time. Combining these results with fast algorithms for uncompressed approximate string matching leads to several efficient algorithms for approximate string matching on grammar-compressed strings without decompression. For instance, we can find all approximate occurrences of a pattern P with at most k errors in time $O(n(\min\{|P|k, k^4 + |P|\} + \log N) + \text{occ})$, where occ is the number of occurrences of P in S. Finally, we generalize our results to navigation and other operations on grammar-compressed ordered trees. All of the above bounds significantly improve the currently best known results. To achieve these bounds, we introduce several new techniques and data structures of independent interest, including a predecessor data structure, two "biased" weighted ancestor data structures, and a compact representation of heavy paths in grammars.

Key words. grammar-based compression, straight-line program, approximate string matching, tree compression

AMS subject classifications. 68P05, 68P30

DOI. 10.1137/130936889

1. Introduction. Modern textual or semistructured databases, e.g., for biological and World-Wide Web data, are huge and are typically stored in compressed form. A query to such databases will typically retrieve only a small portion of the data. This presents several challenges: how to query the compressed data directly and efficiently, without the need for additional data structures (which can be many times larger than the compressed data), and how to retrieve the answers to the queries. In many

practical cases, the naive approach of first decompressing the entire database and then processing it is completely unacceptable—for instance, XML data compresses by an order of magnitude on disk [25] but *expands* by an order of magnitude when represented in-memory [22]; as we will shortly see, this approach is very problematic from an asymptotic perspective as well. Instead we want to support this functionality directly on the compressed data.

We focus on two data types, *strings* and *ordered trees*, and consider the former first. Let S be a string of length N from an alphabet Σ, given in a compressed representation $\mathcal{S}$ of size n. The *random access problem* is to compactly represent $\mathcal{S}$ while supporting fast random access queries, that is, given an index i, $1 \leq i \leq N$, to report $S[i]$. More generally, we want to support *substring decompression*, that is, given a pair of indices i and j, $1 \leq i \leq j \leq N$, report the substring $S[i] \cdots S[j]$. The goal is to use little space for the representation of $\mathcal{S}$ while supporting fast random access and substring decompression. Once we obtain an efficient substring decompression method, it can also serve as a basis for a compressed version of classical pattern matching. For example, given an (uncompressed) pattern string P and $\mathcal{S}$, the *compressed pattern matching problem* is to find all occurrences of P within S more efficiently than to naively decompress $\mathcal{S}$ into S and then search for P in S. An important variant of the pattern matching problem is when we allow approximate matching (i.e., when P is allowed to appear in S with some errors).
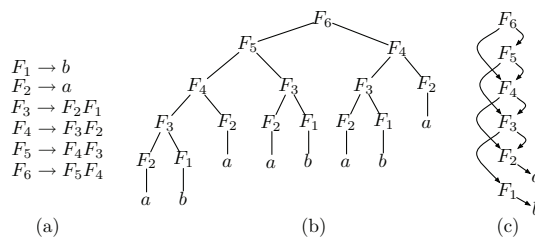

FIG. 1. (a) *A context-free grammar generating the string abaababa.* (b) *The corresponding parse tree.* (c) *The acyclic graph defined by the grammar.*

We consider these problems in the context of *grammar-based compression*, where one replaces a long string by a small context-free grammar (CFG) that generates this string (and this string only). We assume without loss of generality that the grammars are in fact *straight-line programs* (SLPs), and so on the right-hand side of each grammar rule there are either exactly two variables or one terminal symbol; see Figure 1(a)). Such grammars capture many popular compression schemes including the Lempel–Ziv family [64, 66, 67], Sequitur [55], run-length encoding, Re-Pair [45], and many more [6, 7, 8, 29, 41, 42, 60, 65]. All of these are or can be transformed into equivalent grammar-based compression schemes with little expansion [17, 57]. In general, the size of the grammar, defined as the total number of symbols in all derivation rules, can be exponentially smaller than the string it generates. From an algorithmic perspective, the properties of compressed data have been used to accelerate the solutions to classical problems on strings including exact pattern matching [4, 39, 46, 48, 61] and approximate pattern matching [3, 9, 12, 14, 16, 21, 34, 38, 39, 47, 53].

We also consider the problem of representing an ordered rooted tree T (of arbitrary degree) with N nodes. We assume that T is represented as a *balanced parenthesis* sequence [50], which is obtained by traversing T in preorder and outputting "(" upon entering a node for the first time, and ")" upon leaving it. We assume that this

balanced parenthesis sequence is given as an SLP of size n, and consider the problem of performing operations on T.

This compression method may seem a little artificial, but it is in fact a powerful technique that captures existing tree compression methods. For example, a popular tree compression method is to represent the tree by the minimal directed acyclic graph (DAG) obtained by sharing identical subtrees, giving a DAG with e edges (see Figure 7 in section 8). We note that this DAG can be represented as an SLP of size $O(e)$ that generates the BP (balanced parenthesis) sequence of the tree. However, the SLP representation of the BP string can be much smaller than the minimal DAG: for example, if T is just a line with N nodes, then DAG compression is completely ineffective, but the BP sequence of T, namely $(^N)^N$, is generated by an SLP of size $O(\log N)$. In fact, recently, Navarro and Pereira [54] implemented a practical variant of this idea and showed its effectiveness for highly repetitive text collections.

Our results. We present new representations of grammar-compressed strings and trees. We consider two models, the *pointer machine* [62] and the *word RAM* (henceforth just RAM) [32]. We further make the assumption that all memory cells can contain $\log N$-bit integers—this many bits are needed just to represent the input to a random access query. Let $\alpha_k(n)$ be the inverse of the kth row of Ackermann's function.[1] For strings, we show the following result.

THEOREM 1.1. *For a CFG $\mathcal{S}$ of size n representing a string of length N we can decompress a substring of length m in time $O(m + \log N)$*

(i) *after $O(n \cdot \alpha_k(n))$ preprocessing time and space, for any fixed k, or*

(ii) *after $O(n)$ preprocessing time and space on the RAM model.*

Next, we show how to combine Theorem 1.1 with any black-box (uncompressed) approximate string matching algorithm to solve the corresponding compressed approximate string matching problem over grammar-compressed strings. We obtain the following connection between classical (uncompressed) and grammar-compressed approximate string matching. Let $t(m)$ and $s(m)$ be the time and space bounds of some (uncompressed) approximate string matching algorithm on strings of lengths $O(m)$, and let occ be the number of occurrences of P in S.

THEOREM 1.2. *Given a CFG $\mathcal{S}$ of size n representing a string of length N and a string P of length m, we can find all approximate occurrences of P in S in time $O(n(m + t(m) + \log N) + \text{occ})$ and*

(i) *in space $O(n \cdot \alpha_k(n) + m + s(m))$ on the pointer machine model, for any fixed k, and*

(ii) *in space $O(n + m + s(m) + \text{occ})$ on the RAM model.*

Coming to the tree representation problem, suppose that nodes of the uncompressed tree T are numbered $1, \ldots, N$ in preorder, and T is represented as an SLP that generates its BP sequence. We are mainly concerned with navigation operations in the tree such as parent(i) and lca(i, j), which return the (preorder) numbers of the node that is the parent of i or the LCA (lowest common ancestor) of i and j, respectively. (A full list of navigation operations can be found in Table 1.) We show the following.

THEOREM 1.3. *Given an SLP of size n that represents the BP sequence of a rooted ordered tree T with N nodes, we can support the navigation operations given in Table 1 in $O(\log N)$ time using*

---

[1]The inverse Ackermann function $\alpha_k(n)$ can be defined by $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$ so that $\alpha_1(n) = n/2$, $\alpha_2(n) = \log n$, $\alpha_3(n) = \log^* n$, $\alpha_4(n) = \log^{**} n$, and so on. Here, $\log^{**} n$ is the number of times the $\log^*$ function is applied to n to produce a constant.

TABLE 1
Navigational operations on an ordered tree.

`parent`(x)	Parent of node x
`first_child`(x)	First child of node x
`last_child`(x)	Last child of node x
`next_sibling`(x)	Next (previous) sibling of node x
(`prev_sibling`(x))	
`depth`(x)	Depth of node x
`level_anc`(x, i)	Ancestor of node x that is i levels above x, for $i \geq 0$
`desc`(x)	Number of descendants (subtree size) of node x
`height`(x)	Returns the height of the subtree rooted at node x
`lca`(x, y)	Returns the lowest common ancestor of the nodes x and y
`left_leaf`(x)	Leftmost (rightmost) leaf of the subtree rooted at node x
(`right_leaf`(x))	
rank$_{\mathrm{PRE/POST}}(x)$	Position of x in the preorder or postorder traversal of the tree
select$_{\mathrm{PRE/POST}}(j)$	jth node in the preorder or postorder traversal of the tree
`level_left`(i)	First (last) node visited in a preorder traversal among all the
(`level_right`(i))	nodes whose depths are i
`level_succ`(x)	Level successor (predecessor) of node x, i.e., the node visited
(`level_pred`(x))	immediately after (before) node x in a preorder traversal
	among all the nodes that are at the same level as node x

(i) $O(n\alpha_k(n))$ *words and preprocessing time on the pointer machine model, for
any fixed k, and*

(ii) $O(n)$ *words and preprocessing time on the RAM model.*

Remark. In the applications discussed above, it is more appropriate to consider
labelled trees [25], where each node is labelled with a character from some alphabet
Σ. A basic operation on the labels is access(i), which returns the symbol associated
with node i. This can be readily implemented in $O(\log N)$ time by SLP-compressing
the string that comprises the labels of T in preorder and using Theorem 1.1. Note
that separately SLP-compressing the tree structure and the labels of T in preorder
cannot be asymptotically worse than SLP-compressing (say) a "labelled" parenthesis
string, obtained by outputting "$($ $_c$" upon entering a node labelled c and "$)$ $_c$" upon
leaving it.

Related work. We now describe how our work relates to existing results.

The random access problem. If we use $O(N)$ space, we can access any character
in constant time by storing S explicitly in an array. Alternatively, we can compute
and store the sizes of strings derived by each grammar symbol in $\mathcal{S}$. This requires
only $O(n)$ space and allows us to simulate a top-down search expanding the gram-
mar's derivation tree in constant time per node. Consequently, a random access takes
time $O(h)$, where h is the height of the derivation tree and can be as large as $\Omega(n)$.
Although any SLP of size n generating a string of length N can be converted into an
SLP with derivation tree height $O(\log N)$ [17, 57], the size of the SLP may increase
to $O(n \log N)$. Thus, the simple top-down traversal either has poor worst-case perfor-
mance or uses nonlinear space. Surprisingly, the only known improvement to the sim-
ple top-down traversal is a recent succinct representation of grammars, due to Claude
and Navarro [19]. They reduce the space from $O(n \log N)$ bits to $O(n \log n) + n \log N$
bits at the cost of increasing the query time to $O(h \log n)$.

The substring decompression problem. Using the simple random access trade-off,
we get an $O(n)$ space solution that supports substring decompression in $O(hm)$ time.
Gasieniec and coworkers [30, 31] showed how to improve the decompression time to
$O(h + m)$ while maintaining $O(n)$ space. The key idea in their solution is to use and

extend the simple top-down traversal to the first character of the substring with a fast traversal of the derivation tree of the remaining substring. Also, the representation of Claude and Navarro [19] supports substring decompression in time $O((h + m) \log n)$.

The compressed pattern matching problem. In approximate pattern matching, we are given two strings P and S and an *error threshold k*. The goal is to find all ending positions of substrings of S that are "within distance k" of P under some metric, e.g., the *edit distance* metric, where the distance is the number of edit operations needed to convert one substring to the other.

In classical (uncompressed) approximate pattern matching, a simple algorithm [59] solves this problem (under edit distance) in $O(Nm)$ time and $O(m)$ space, where N and m are the lengths of S and P, respectively. Several improvements of this result are known (see, e.g., [52]). Two well-known improvements for small values of k are the $O(Nk)$ time algorithm of Landau and Vishkin [44] and the $O(Nk^4/m + N)$ time algorithm of Cole and Hariharan [20]. Both of these can be implemented in $O(m)$ space. The use of compression has led to many speedups using various compression schemes [3, 9, 12, 14, 16, 21, 34, 38, 39, 40, 47, 53]. The work most closely related to ours is approximate pattern matching for LZ78 and LZW compressed strings [12, 38, 53], which can be solved in time $O(n(\min\{mk, k^4 + m\}) + \text{occ})$ [12], where n is the compressed length under the LZ compression.

Theorem 1.2 gives us the first nontrivial algorithms for approximate pattern matching over any grammar-compressed string. For instance, if we plug the Landau–Vishkin [44] or Cole–Hariharan [20] algorithms into Theorem 1.2(i), we obtain an algorithm with $O(n(\min\{mk, k^4 + m\} + \log N) + \text{occ})$ time and $O(n \cdot \alpha_k(n) + m + \text{occ})$ space. Note that any algorithm (not only the above two) and any distance metric (not only edit distance) can be applied to Theorem 1.2. For example, under the Hamming distance measure we can combine our algorithm with a fast algorithm for the (uncompressed) approximate string matching problem for the Hamming distance measure [5].

Tree compression. There is a long history of tree compression algorithms, but there appears to be little work on rapidly navigating the compressed representation without decompression. In particular, the DAG compression approach has recently been applied successfully to compress XML documents [13, 15]; also note that this representation aids the matching of XPath patterns, but their algorithm partially decompresses the DAG. Indeed [15, p. 468] specifically mentions the problem of navigating the XML tree without decompressing the DAG and presents algorithms whose running time is linear in the grammar size for randomly accessing the nodes of the tree. Jansson, Sadakane, and Sung [37] give an ordered tree representation that supports a wide variety of navigational operations on a compressed ordered tree. However, their compression method is relatively weak—it is based solely on the degree distribution of the nodes in the tree—and cannot fully exploit repeated substructure in trees.

Overview. Before diving into technical details, we give an outline of the paper and of the new techniques and data structures that we introduce and believe to be of independent interest. We first focus on the string random access problem. Let $\mathcal{S}$ be an SLP of size n representing a string of length N. We begin in section 2 by defining a forest H of size n that represents the heavy paths [33] in the parse tree of $\mathcal{S}$. We then combine the forest H with an existing *weighted ancestor* data structure,[2] leading

---

[2]A weighted ancestor query (v, p) asks for the lowest ancestor of v whose weighted distance from v is at least p.

to a first solution with $O(\log N \log \log N)$ access time and linear space (Lemma 2.1). The main part of the paper focuses on reducing the random access time to $O(\log N)$.

In section 3, we observe that it is better to replace the doubly logarithmic weighted ancestor search in Lemma 2.1 by a (logarithmic) *biased* ancestor search. In a biased search, we want to find the predecessor of a given integer p in a set of integers $0 = l_0 < l_1 < \cdots < l_k = U$, in $O(\log(U/x))$ time, where $x = |\text{successor}(p) - \text{predecessor}(p)|$.[3] Using biased search, the $O(\log N)$ predecessor queries on H add up to just $O(\log N)$ time overall. Our main technical contribution is to design two new space-efficient data structures that perform biased searches on sets defined by any path from a node $u \in H$ to the root of u's tree. In section 3 we describe the central building block of the first data structure—the *interval-biased search tree*, which is a new, simple linear-time constructable, linear space, biased search data structure. We cannot directly use this data structure on every node-to-root path in H, since that would take $O(n^2)$ preprocessing time and space. In section 4 we first apply a heavy-path decomposition to H itself and navigate between these paths using weighted ancestor queries on a related tree L. This reduces the preprocessing time to $O(n \log n)$. To further reduce the preprocessing, we partition L into disjoint trees in the spirit of Alstrup, Husfeldt, and Rauhe [2]. One of these trees has $O(n/\log n)$ leaves and can be preprocessed using the solution above. The other trees all have $O(\log n)$ leaves, and we handle them recursively. However, before we can recurse on these trees they are modified so that each has $O(\log n)$ vertices (rather than leaves). This is done by another type of path decomposition (i.e., not a heavy-path decomposition), called *branching decomposition* of L. By carefully choosing the sizes of the recursive problems, we get Theorem 1.1(i) (for the case $m = 1$).

For the RAM model, in section 5, we generalize *biased skip lists* [10] to *biased skip trees*, where every path from a node $u \in H$ to u's root is a biased skip list, giving the required time complexity. While a biased skip list takes linear space [35], a biased skip tree may have $\Omega(n \log N)$ pointers and hence nonlinear space, since in a biased skip list, "overgrown" nodes (those with many more pointers than justified by their weight) are amortized over those ancestors which have an appropriate number of pointers. When used in H, however, the parent of an overgrown node may have many overgrown children, all sharing the same set of ancestors, and the amortization fails. We note that no node will have more than $O(\log N)$ pointers, and we use a sequence of $O(\log N)$ *succinct* trees [51] of $O(|H|) = O(n)$ bits each to represent the skip list pointers, using $O(n \log N)$ bits or $O(n)$ words in all. These succinct trees support in $O(1)$ time a new *colored ancestor query*—a natural operation that may find other uses—using which we are able to follow skip list pointers in $O(1)$ time, giving the bounds of Theorem 1.1(ii) (for the case $m = 1$).

We extend both random access solutions to the substring decompression in section 6, and in section 7 we combine our substring decompression result with a technique of [12] to obtain an algorithm for approximate matching grammar-compressed strings (giving the bounds of Theorem 1.2). The algorithm computes the approximate occurrences of the pattern in a single bottom-up traversal of the grammar. At each step we use the substring decompression algorithm to decode a relevant small portion of the string, thus avoiding a full decompression.

Finally, in section 8, we describe the differences between the random access op-

---

[3]Note that we need a slightly different property than so-called *optimum* binary search trees [43, 49]—we do not want to minimize the total external path length but rather ensure that *each* item is at its ideal depth, as in [11].

eration in trees and that in strings. The structure of the trees is represented by a balanced parenthesis string, which is assumed to be stored in a compressed representation as an SLP. Although the basic approach is very much the same as that for random access on strings, the key new operation that we support on SLP-encoded balanced parenthesis strings is *excess search* [51, 58]. Along the way, we show how to extend the random access representation to support rank and select operations [36] on a binary string encoded as an SLP.

2. Fast random access in linear space and $O(\log N \log \log N)$ access time. In the rest of the paper, we let $\mathcal{S}$ denote an SLP of size n representing a string of length N and let T be the corresponding parse tree (see Figure 1(b)). In this section we present an $O(n)$ space representation of $\mathcal{S}$ that supports random access in $O(\log N \log \log N)$ time, which also introduces the general framework. To achieve this we partition $\mathcal{S}$ into disjoint paths according to a *heavy-path decomposition* [33], and from these form the *heavy-path suffix forest*, which is of size $O(n)$.

Heavy-path decompositions. Similar to Harel and Tarjan [33], we define the *heavy-path decomposition* of the parse tree T as follows. For each node v define $T(v)$ to be the subtree rooted at v, and let $\text{size}(v)$ be the number of descendant leaves of v. We classify each node in T as either *heavy* or *light* based upon $\text{size}(v)$.[4] The root is light. For each internal node v we pick a child of maximum size and classify it as heavy. The heavy child of v is denoted $\text{heavy}(v)$. The remaining children are light. An edge to a light child is a *light edge*, and an edge to a heavy child is a *heavy edge*. Removing the light edges, we partition T into *heavy paths*. A *heavy path suffix* is a simple path $v_1, \ldots, v_k$ from a node v_1 to a leaf in $T(v_1)$ such that $v_{i+1} = \text{heavy}(v_i)$ for $i = 1, \ldots, k - 1$. If u is a light child of v, then $\text{size}(u) \leq \text{size}(v)/2$, since otherwise u would be heavy. Consequently, the number of light edges on a path from the root to a leaf is at most $O(\log N)$ [33].

We extend heavy-path decomposition of trees to SLPs in a straightforward manner. We consider each grammar variable v as a node in the directed acyclic graph defined by the grammar (see Figure 1(c)). For a node v in $\mathcal{S}$ let $S(v)$ be the substring induced by the parse tree rooted at v and define the size of v to be the length of $S(v)$. We define the heavy paths in $\mathcal{S}$ as in T from the size of each node. Since the size of a node v in $\mathcal{S}$ is the number of leaves in $T(v)$, the heavy paths are well defined, and we may reuse all of the terminology for trees on SLPs. In a single $O(n)$ time bottom-up traversal of $\mathcal{S}$ we can compute the sizes of all nodes and hence the heavy-path decomposition of $\mathcal{S}$.

Fast random access in linear space. Our data structure represents the following information for each heavy path suffix $v_1, \ldots, v_k$ in $\mathcal{S}$:
- The length $\text{size}(v_1)$ of the string $S(v_1)$.
- The index z of v_k in the left-to-right order of the leaves in $T(v_1)$ and the character $S(v_1)[z]$.
- A predecessor data structure for the *left size sequence* $l_0, l_1, \ldots, l_k$, where l_i is the sum of 1 plus the sizes of the *light-left children* (i.e., each light node that is a left child of its parent) of the first i nodes in the heavy-path suffix.
- A predecessor data structure for the *right size sequence* $r_0, \ldots, r_k$, where r_i is the sum of 1 plus the sizes of the *light-right children* of the first i nodes in the heavy-path suffix.

---

[4]Note that our definition of heavy paths is slightly different from the usual one. We construct our heavy paths according to the number of leaves of the subtrees and not the total number of nodes.
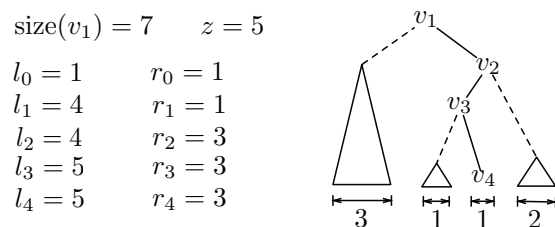
$$\text{size}(v_1) = 7 \qquad z = 5$$

$$
\begin{array}{ll}
l_0 = 1 & r_0 = 1 \\
l_1 = 4 & r_1 = 1 \\
l_2 = 4 & r_2 = 3 \\
l_3 = 5 & r_3 = 3 \\
l_4 = 5 & r_4 = 3
\end{array}
$$

FIG. 2. *Ancestor search in H. The left- and right size sequences for a heavy path suffix v_1, v_2, v_3, v_4. The dotted edges are to light subtrees, and the numbers at the bottom are subtree sizes. A search for $p = 5$ returns the stored character for $S(v_1)[z]$. A search for $p = 4$ computes the predecessor l_2 of 4 in the left size sequence. The search continues in the left subtree of v_3 for position $p - l_2 + 1 = 4 - 4 + 1 = 1$. A search for $p = 6$ computes the predecessor r_1 of $7 - 6 = 1$ in the right size sequence. The search continues in the right subtree of v_2 for position $p - z = 6 - 5 = 1$.*

With this information we perform a top-down search of T as follows. Suppose that we have reached node v_1 with heavy-path suffix $v_1, \ldots, v_k$ and our goal is to access the character $S(v_1)[p]$. We then compare p with the index z of v_k. There are three cases (see Figure 2 for an example):

1. If $p = z$, we report the stored character $S(v_1)[z]$ and end the search.
2. If $p < z$, we compute the predecessor l_i of p in the left size sequence. We continue the top-down search from the left child u of v_{i+1}. The position of p in $T(u)$ is $p - l_i + 1$.
3. If $p > z$, we compute the predecessor r_i of $\text{size}(v_1) - p$ in the right size sequence. We continue the top-down search from the right child u of v_{i+1}. The position of p in $T(u)$ is $p - (z + \sum_{j=i+2}^{k} \text{size}(v_j))$. (Note that we can compute the sum in constant time as $r_k - r_{i+2}$.)

The total length of all heavy-path suffixes is $O(n^2)$, thus making it unattractive to treat each suffix independently. We show how to compactly represent all of the predecessor data structures from the algorithm of the previous section in $O(n)$ space and introduce the *heavy-path suffix forest H of $\mathcal{S}$*. The nodes of H are the nodes of S, and a node u is the parent of v in H iff u is the heavy child of v in $\mathcal{S}$. Thus, a heavy path suffix $v_1, \ldots, v_k$ in $\mathcal{S}$ is a sequence of ancestors from v_1 in H. We label the edge from v to its parent u by a left weight, and right weight is defined as follows. If u is the left child of v in $\mathcal{S}$, the left weight is 0 and the right weight is $\text{size}(v')$, where v' is the right child of v. Otherwise, the right weight is 0 and the left weight is $\text{size}(v')$, where v' is the left child of v. Heavy-path suffixes in $\mathcal{S}$ consist of unique nodes, and therefore H is a forest. A heavy-path suffix in $\mathcal{S}$ ends at one of $|\Sigma|$ leaves in $\mathcal{S}$, and therefore H consists of up to $|\Sigma|$ trees, each rooted at a unique character of Σ. The total size of H is $O(n)$, and we may easily compute it from the heavy-path decomposition of $\mathcal{S}$ in $O(n)$ time. An example of a heavy-path decomposition and corresponding heavy-path suffix forest is shown in Figure 3.

A predecessor query on a left size sequence and right size sequence of a heavy-path suffix $v_1, \ldots, v_k$ is now equivalent to a *weighted ancestor query* on the left weights and right weights of H, respectively. Farach and Muthukrishnan [24] showed how to support weighted ancestor queries in $O(\log \log N)$ time after $O(n)$ space and preprocessing time. Hence, if we plug this into our algorithm, we obtain $O(\log N \log \log N)$ query time with $O(n)$ preprocessing time and space. In summary, we have the following result.
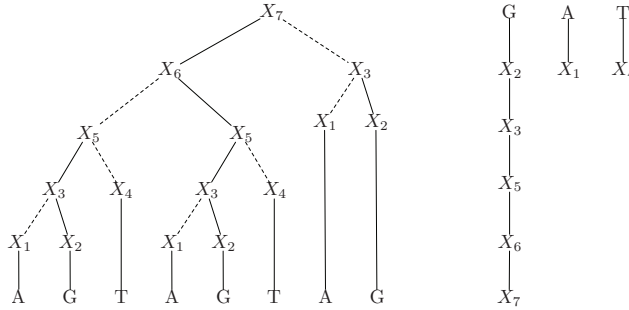
FIG. 3. *Heavy-path decomposition of a derivation tree and corresponding heavy-path suffix forest. Dashed edges in the derivation tree are light, and solid edges are heavy. In this case, the heavy-path suffix forest consists of $|\Sigma| = 3$ paths.*

LEMMA 2.1. *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N \log \log N)$ after $O(n)$ preprocessing time and space.*

3. Interval-biased search trees—Removing the $O(\log \log N)$ factor. In this section we reduce the $O(\log N \log \log N)$ random access time on an SLP $\mathcal{S}$ in Lemma 2.1 to $O(\log N)$. Recall that $O(\log N \log \log N)$ was a result of performing $O(\log N)$ predecessor(p) queries, each in $O(\log \log N)$ time. In this section, we introduce a new predecessor data structure—the *interval-biased search tree*. Each predecessor(p) query on this data structure requires $O\big(\log \frac{U}{x}\big)$ time, where $x = $ successor(p) – predecessor(p) and U is the universe. The same result can be achieved using a general biased search tree (see, e.g., [11]), with weights corresponding to the difference between pairs of consecutive points. If we allow non–pointer-based techniques, it is even possible to achieve $O\big(\log \log \frac{U}{x}\big)$ [56]. Our new construction specialized to intervals is quite simple and may be of independent interest.

To see the advantage of $O\big(\log \frac{U}{x}\big)$ predecessor queries over $O(\log \log N)$, suppose that after performing the predecessor query on the first heavy path of T we discover that the next heavy path to search is the heavy-path suffix originating in node u. This means that the first predecessor query takes $O\big(\log \frac{N}{|S(u)|}\big)$ time. Furthermore, the elements in u's left size sequence (or right size sequence) are all from a universe $\{0, 1, \ldots, |S(u)|\}$. Therefore, the second predecessor query takes $O\big(\log \frac{|S(u)|}{x}\big)$, where $x = |S(u')|$ for some node u' in $T(u)$. The first two predecessor queries thus require time $O\big(\log \frac{N}{|S(u)|} + \log \frac{|S(u)|}{x}\big) = O\big(\log \frac{N}{x}\big)$. The time required for all $O(\log N)$ predecessor queries telescopes similarly for a total of $O(\log N)$.

We next show how to construct an interval-biased search tree in linear time and space. Simply using this tree on each heavy-path suffix of $\mathcal{S}$ already results in a structure for an SLP $\mathcal{S}$ of size n representing a string of length N that can support random access in time $O(\log N)$ after $O(n^2)$ preprocessing time and space.

A description of the tree. We now define the interval-biased search tree associated with $\hat{n}$ integers $l_1 \leq \cdots \leq l_{\hat{n}}$ from a universe $\{0, 1, \ldots, \hat{N}\}$. For simplicity, we add the elements $l_0 = 0$ and $l_{\hat{n}+1} = \hat{N}$. The interval-biased search tree is a binary tree that stores the intervals $[l_0, l_1], [l_1, l_2], \ldots, [l_{\hat{n}}, l_{\hat{n}+1}]$ with a single interval in each node. The tree is described recursively:

   1. Let i be such that $(l_{\hat{n}+1} - l_0)/2 \in [l_i, l_{i+1}]$. The root of the tree stores the

interval $[l_i, l_{i+1}]$.

2. The left child of the root is the interval-biased search tree storing the intervals $[l_0, l_1], \ldots, [l_{i-1}, l_i]$, and the right child is the interval-biased search tree storing the intervals $[l_{i+1}, l_{i+2}], \ldots, [l_{\hat{n}}, l_{\hat{n}+1}]$.

Notice that the total length of the intervals for the left and right subtrees is at most $\hat{N}/2$. Inductively, the total size of the intervals at depth j is at most $\hat{N}/2^j$. Hence, any interval of length x such that $\hat{N}/2^j \leq x \leq \hat{N}/2^{j+1}$ must be stored in a node of depth at most j.

When we search the tree for a query p and reach a node corresponding to the interval $[l_i, l_{i+1}]$, we compare p with l_i and l_{i+1}. If $l_i \leq p \leq l_{i+1}$, then we return l_i as the predecessor. If $p < l_i$ (resp., $p > l_{i+1}$), we continue the search in the left child (resp., right child). Since the interval $[l_i, l_{i+1}]$ of length $x = l_{i+1} - l_i$ such that $\hat{N}/2^j \leq x \leq \hat{N}/2^{j+1}$ is stored in a node of depth at most j, the search terminates at a node of depth at most j. The query time is thus $j \leq 1 + \log \frac{\hat{N}}{x} = O\big(\log \frac{\hat{N}}{x}\big)$, which is exactly what we desire as $x = \text{successor}(p) - \text{predecessor}(p)$. We now give an $O(\hat{n})$ time and space algorithm for constructing the tree.

A linear-time construction of the tree. We describe an $O(\hat{n})$ time and space top-down construction of the interval-biased search tree storing the intervals $[l_j, l_{j+1}], \ldots, [l_k, l_{k+1}]$. We focus on finding the interval $[l_i, l_{i+1}]$ to be stored in its root. The rest of the tree is constructed recursively so that the left child is a tree storing the intervals $[l_j, l_{j+1}], \ldots, [l_{i-1}, l_i]$ and the right child is a tree storing the intervals $[l_{i+1}, l_{i+2}], \ldots, [l_k, l_{k+1}]$.

We are looking for an interval $[l_i, l_{i+1}]$ such that i is the largest value where $l_i \leq (l_{k+1} + l_j)/2$ holds. We can find this interval in $O(\log(k - j))$ time by doing a binary search for $(l_{k+1} + l_j)/2$ in the subarray $l_j, l_{j+1}, \ldots, l_{k+1}$. However, notice that we are not guaranteed that $[l_i, l_{i+1}]$ partitions the intervals in the middle. In other words, $i - j$ can be much larger than $k - i$ and vice versa. This means that the total time complexity of all the binary searches we do while constructing the entire tree can amount to $O(n \log n)$, and we want $O(n)$. To overcome this, notice that we can find $[l_i, l_{i+1}]$ in $\min\{\log(i - j), \log(k - i)\}$ time if we use a doubling search from both sides of the subarray, that is, if prior to the binary search, we narrow the search space by doing a parallel scan of the elements $l_j, l_{j+2}, l_{j+4}, l_{j+8}, \ldots$ and $l_k, l_{k-2}, l_{k-4}, l_{k-8}, \ldots$. This turns out to be crucial for achieving $O(n)$ total construction time, as we now show.

To verify the total construction time, we need to bound the total time required for all the binary searches. Let $T(\hat{n})$ denote the time complexity of all the binary searches; then $T(\hat{n}) = T(i) + T(\hat{n} - i) + \min\{\log i, \log(\hat{n} - i)\}$ for some i. Setting $d = \min\{i, \hat{n} - i\} \leq \hat{n}/2$, we get that $T(\hat{n}) = T(d) + T(\hat{n} - d) + \log d$ for some $d \leq \hat{n}/2$, which is $O(\hat{n})$.[5]

Final tuning. We need one last important property of the interval-biased search tree. Suppose that right before doing a predecessor(p) query we know that $p > l_k$ for some k. We can reduce the query time to $O\big(\log \frac{\hat{N} - l_k}{x}\big)$ by computing for each node its lowest common ancestor with the node $[l_{\hat{n}}, l_{\hat{n}+1}]$, in a single traversal of the tree. Then, when searching for p, we can start the search in the lowest common ancestor of $[l_k, l_{k+1}]$ and $[l_{\hat{n}}, l_{\hat{n}+1}]$ in the interval-biased search tree. We will need this property

---

[5]By an inductive assumption that $T(\hat{n}) < 2\hat{n} - \log \hat{n} - 2$ we get that $T(\hat{n})$ is at most $2d - \log d - 2 + 2(\hat{n} - d) - \log(\hat{n} - d) - 2 + \log d = 2\hat{n} - \log(\hat{n} - d) - 4$, which is at most $2\hat{n} - \log \hat{n} - 3$ since $d \leq \hat{n}/2$.
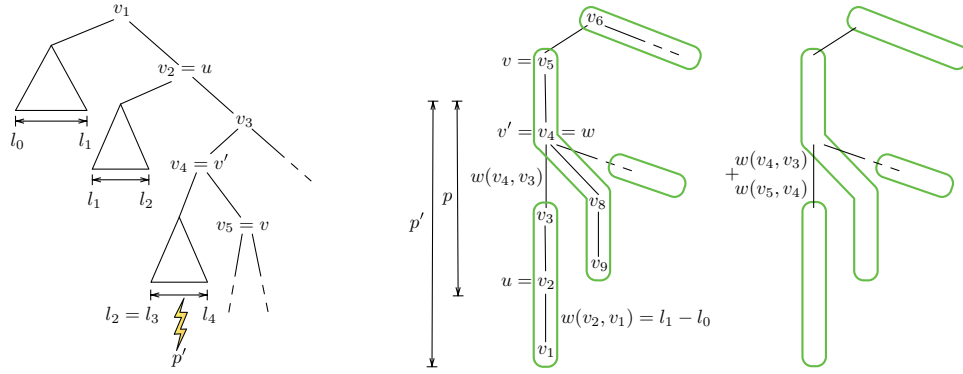
FIG. 4. *The parse tree T of an SLP (left), the* heavy-path suffix forest H *(middle), and the light representation L of H (right). The heavy-path decomposition of H is marked in the middle and right panels (in green) and defines the vertex set of L.*

to efficiently perform biased searches on suffixes of paths in the following sections. In summary, we have the following result.

LEMMA 3.1. *Given a sequence of $\hat{n}$ integers $l_1 \leq \cdots \leq l_{\hat{n}}$ from a universe $\{0, 1, \ldots, U\}$, we can support predecessor search for a point p in time $O(\log \frac{U}{x})$, where $x = \mathrm{successor}(p) - \mathrm{predecessor}(p)$, and linear space. Furthermore, if we have a pointer to l_k and $p > l_k$, we can support predecessor search in time $O\big(\log \frac{U - l_k}{x}\big)$.*

4. Closing the time-space trade-offs for random access. In this section we will use the interval-biased search tree to achieve $O(\log N)$ random access time but near-linear space usage and preprocessing time (instead of $O(n^2)$ as in Lemma 2.1). We design a novel *weighted ancestor* data structure on H via a heavy-path decomposition of H itself. We use interval-biased search trees for each heavy path P in this decomposition: one each for the left- and right size sequences. It is easy to see that the total size of all these interval-biased search trees is $O(n)$. We focus on queries of the left size sequence; the right size sequence is handled similarly.

Let P be a heavy path in the decomposition, let v be a vertex on this path, and let $w(v, v')$ be the weight of the edge between v and its child v'. We denote by $b(v)$ the weight of the part of P below v, and by $t(v)$ the weight above v. As an example, consider the green heavy path $P = (v_5\text{-}v_4\text{-}v_8\text{-}v_9)$ in Figure 4; then $b(v_4) = w(v_4, v_8) + w(v_8, v_9)$ and $t(v_4) = w(v_5, v_4)$. In general, if $P = (v_k\text{-}v_{k-1}\text{-}\cdots\text{-}v_1)$, then v_1 is a leaf in H and $b(v_{i+1})$ is the ith element in P's predecessor data structure. The $b(\cdot)$ and $t(\cdot)$ values of all vertices can easily be computed in $O(n)$ time.

Recall that, given any vertex u in H and any $0 \leq p \leq N$, we need to be able to find the lowest ancestor v of u whose weighted distance from u is at least p. If we want the total random access time to be $O(\log N)$, then finding v should be done in $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time, where v' is the child of v which is also an ancestor of u. If both u and v are on the same heavy path P in the decomposition, a single predecessor(p') query on P would indeed find v in $O\big(\log \frac{t(u)}{w(v,v')}\big) = O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time, where $p' = p + b(u)$. This follows from the property we described at the end of section 3.

The problem is thus to locate v when, in the decomposition of H, v is on the heavy path P but u is not. To do so, we first locate a vertex w that both is an ancestor of u and belongs to P. Once w is found, if its weighted distance from u is greater than p,

then $v = w$. Otherwise, a single predecessor(p'') query on P finds v in $O\big(\log \frac{t(w)}{w(v,v')}\big)$ time, which is $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ since $t(w) \leq |S(u)|$. Here, $p'' = p - \text{weight}(\text{path from } u \text{ to } w \text{ in } H) + b(w)$. We are therefore left with only the problem of finding w and the weight of the path from u to w.

A light representation of heavy paths. In order to navigate from u up to w we introduce the *light representation* L of H. Intuitively, L is a (nonbinary) tree that captures the light edges in the heavy-path decomposition of H. Every path P in the decomposition of H corresponds to a single vertex P in L, and every light edge in the decomposition of H corresponds to an edge in L. If a light edge e in H connects a vertex w with its child, then the weight of the corresponding edge in L is the original weight of e plus $t(w)$. (See the edge of weight $w(v_4, v_3) + w(v_5, v_4)$ in Figure 4.)

The problem of locating w in H now translates to a weighted ancestor query on L. Indeed, if u belongs to a heavy path P', then P' is also a vertex in L, and locating w translates to finding the lowest ancestor of P' in L whose weighted distance from P' is at least $p - t(u)$. As a weighted ancestor data structure on L would be too costly, we utilize the important fact that the height of L is only $O(\log n)$—the edges of L correspond to light edges of H—and construct, for every root-to-leaf path in L, an interval-biased search tree as its predecessor data structure. The total time and space for constructing these data structures is $O(n \log n)$. A query for finding the ancestor of P' in L whose weighted distance from P' is at least $p - t(u)$ can then be done in $O\big(\log \frac{|S(u)|}{t(w)}\big)$ time. This is $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ as $w(v, v') \leq t(w)$. We summarize this with the following lemma.

LEMMA 4.1. *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N)$ after $O(n \log n)$ preprocessing time and space.*

As noted in the introduction, the further reduction to $O(n\alpha_k(n))$ space and preprocessing time is achieved through a further decomposition of L. Intuitively, we partition L into disjoint trees in the spirit of Alstrup, Husfeldt, and Rauhe [2]. One of these trees has $O(n/\log n)$ leaves and can be preprocessed using the solution above. The other trees all have $O(\log n)$ leaves, and we want to handle them recursively. However, for the recursion to work we will need to modify these trees so that each has $O(\log n)$ vertices (rather than leaves). As described in the following subsection, this is done by another type of path decomposition—a *branching decomposition*.

An inverse-Ackerman-type bound. We have just seen that after $O(n \log n)$ preprocessing we can support random access in $O(\log N)$ time. This superlinear preprocessing originates in the $O(n \log n)$-sized data structure that we construct on L for $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$ time weighted ancestor queries. We now turn to reducing the preprocessing to be arbitrarily close to linear by recursively shrinking the size of this weighted ancestor data structure on L.

In order to do so, we perform a decomposition of L that was originally introduced by Alstrup, Husfeldt, and Rauhe [2] for solving the the *marked ancestor* problem: Given the rooted tree L of n nodes, for every maximally high node whose subtree contains no more than $\log n$ *leaves*, we designate the subtree rooted at this node a *bottom tree*. Nodes not in a bottom tree make up the *top tree*. It is easy to show that the top tree has at most $n/\log n$ leaves and that this decomposition can be done in linear time.

Notice that we can afford to construct, for every root-to-leaf path *in the top tree*, an interval-biased search tree as its predecessor data structure. This is because there will be only $n/\log n$ such data structures and each is of size $\text{height}(L) =$

$O(\log n)$. In this way, a weighted ancestor query that originates in a top tree node takes $O\left(\log \frac{|S(u)|}{w(v,v')}\right)$ time, as required. The problem is therefore handling queries originating in bottom trees.

To handle such queries, we would like to recursively apply our $O(n \log n)$ weighted ancestor data structure on each one of the bottom trees. This would work nicely if the number of *nodes* in a bottom tree were $O(\log n)$. Unfortunately, we only know this about the number of its *leaves*. We therefore use a *branching representation B* for each bottom tree. The number of *nodes* in the representation B is indeed $\log n$ and is defined as follows.

We partition a bottom tree into disjoint paths according to the following rule: A node v belongs to the same path as its child unless v is a branching node (has more than one child). We associate each path P in this decomposition with a unique interval-biased search tree as its predecessor's data structure. The *branching representation B* is defined as follows. Every path P corresponds to a single node in B. An edge e connecting path P' with its parent-path P corresponds to an edge in B whose weight is e's original weight plus the total weighted length of the path P' (see Figure 5).
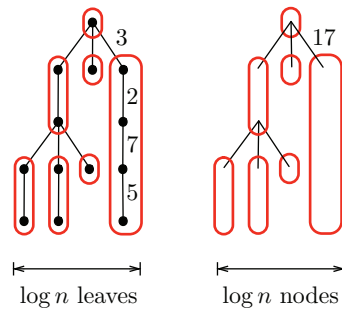

FIG. 5. *A bottom tree and its branching representation B. On the left is some* bottom tree—a *weighted tree with* $\log n$ *leaves. The bottom tree can be decomposed into* $\log n$ *paths (marked in red), each with at most one branching node. Replacing each such path with a single node, we get the* branching representation B*, depicted on the right. The edge-weight* 17 *is obtained by adding the original weight* 3 *to the weighted path* $2 + 7 + 5$.

Each internal node in B has at least two children, and therefore the number of nodes in B is $O(\log n)$. Furthermore, similarly to section 3, our only remaining problem is weighted ancestor queries on B. Once the correct node is found in B, we can query the interval-biased search tree of its corresponding path in L in $O\left(\log \frac{|S(u)|}{w(v,v')}\right)$ time, as required.

Now that we can capture a bottom tree with its branching representation B of logarithmic size, we could simply use our $O(n \log n)$ weighted ancestor data structure on every B. This would require an $O(\log n \log \log n)$ time construction for each one of the $n/\log n$ bottom trees for a total of $O(n \log \log n)$ construction time. In addition, every bottom tree node v stores its weighted distance $d(v)$ from the root of its bottom tree. After this preprocessing, upon query v, we first check $d(v)$ to see whether the target node is in the bottom tree or the top tree. Then, a single predecessor query on the (bottom or top) tree takes $O\left(\log \frac{|S(u)|}{w(v,v')}\right)$ time, as required.

It follows that we can now support random access on an SLP in time $O(\log N)$ after only $O(n \log \log n)$ preprocessing. In a similar manner we can use this $O(n \log \log n)$ preprocessing recursively on every B to obtain an $O(n \log \log \log n)$ solution. Conse-

quently, we can reduce the preprocessing to $O(n \log^* n)$ while maintaining $O(\log N)$ random access. Notice that if we do this naively, then the query time increases by a $\log^* n$ factor due to the $\log^* n$ $d(v)$ values we have to check. To avoid this, we simply use an interval-biased search tree for every root-to-leaf path of $\log^* n$ $d(v)$ values. This requires only an additional $O(n \log^* n)$ preprocessing, and the entire query remains $O\big(\log \frac{|S(u)|}{w(v,v')}\big)$.

Finally, we note that choosing the recursive sizes more carefully (in the spirit of [1, 18]) can reduce the $\log^* n$ factor down to $\alpha_k(n)$ for any fixed k. This gives the following theorem.

THEOREM 4.2. *For an SLP $\mathcal{S}$ of size n representing a string of length N we can support random access in time $O(\log N)$ after $O(n \cdot \alpha_k(n))$ preprocessing time and space for any fixed k on the pointer machine model.*

5. Biased skip trees. In this section we give an alternate representation of the heavy-path suffix forest H, which supports the "biased" predecessor search of the biased interval search tree; the space and preprocessing are both $O(n)$, but the data structure uses the more powerful word RAM model with word size $O(\log N)$ bits. For convenience of description, the predecessor search is expressed a little differently: suppose that we aim to access the pth symbol of $S(v)$ for some node v, and suppose that u is an ancestor of v in H (i.e., u is a heavy descendant of v in the parse tree); assume as previously that the desired symbol is not the symbol associated with the root of the tree in which v is found. We say that a *test* at u is "true" if the desired symbol is in u's heavy child, and "false" otherwise; this test is performed in $O(1)$ time by storing l and r values as before. Our objective is to find the lowest ancestor u in H of v such that the test at u is "false"; this search should take $O(\log(W_v/w_u) + 1)$ time, where for all nodes $u \in H$, $w_u = \text{size}(u')$, where u' is the light child of u, and $W_v = \text{size}(v)$.

Our solution uses a static version of biased skip lists [10], generalized to trees. The initial objective is to assign a nonnegative integral *color* c_v to each node in $v \in H$, and there is a (logical) unidirectional linked list that points up the tree such that all nodes on a leaf-to-root path whose color is at least c are linked together by a series of *color-c pointers*. We defer the implementation of color-c pointers to later, but note here only that we can follow a pointer in $O(1)$ time.

The biased search starting at a node v will proceed essentially as in a skip list. Let c_v^{max} denote the maximum color of any ancestor of v, and c^{max} the maximum color of any node in H. The search first tests v—if the answer is false, we are done; otherwise, we set $c = c_v^{max}$, set the current node to v, and suppose $v' = nca(v, c)$, where $nca(v, c)$ denotes the lowest ancestor of v in the tree whose color is $\geq c$. We test at v'; if the outcome is true, then we set the current node to v'; otherwise we check that v' is not the final answer by testing the appropriate child of v'. If v' is not the final answer, then we set $c = c - 1$ and continue.

We now describe how we select the colors of the nodes in T. For any node v, denote the *rank* of v to be $r_v = \lfloor \log_2 w_v \rfloor + 1$. We perform a preorder traversal of each tree in H. When visiting v, we initially set $c_v = r_v$. Then, while the nearest ancestor of v with color greater than or equal to c_v has color exactly c_v, we increment c_v by one (see Figure 6 for an example). We now show the following result.

LEMMA 5.1.
(1) *For $1 \leq i \leq c^{max} - 1$, between any two consecutive nodes of color i there is a node of color greater than i; there is exactly one node of color c^{max}.*
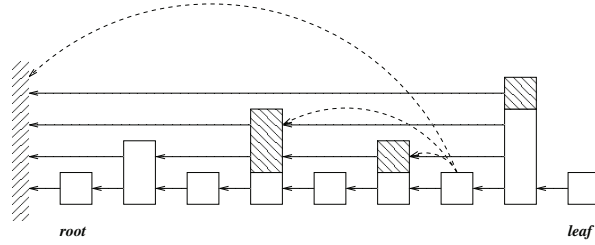(2) *$c_v^{max} \leq 1 + \log_2 W_v$; $c^{max} \leq 1 + \log_2 N$.*

FIG. 6. *Diagram showing the colors assigned to a sequence of vertices with ranks 1 (root), 2, 1, 1, 1, 1, 1, 3, 1 (leaf). The unshaded portion of the tower of a vertex represents its rank; the shaded portion is the "additional" pointers added by the algorithm. Solid arrows show explicit color-c pointers that would be stored in a biased skip list; dotted arrows show examples of pointers that are available implicitly through the nca operation.*

(3) *For any vertex v and ancestor u of v, $c_v^{max} - c_u = O(\log(W_v/w_u))$.*

Proof. Point (1) follows by construction. For (2) and (3), consider any path in H from a node v to the root, and, as in [10], define $N_i = |\{u \in anc(v) : r_u = i\}|$ and $N_i' = |\{u \in anc(v) : r_u \leq i \text{ and } c_u \geq i\}|$, where $anc(x)$ denotes the set of all ancestors of node x. We now show that $N_{i+1}' \leq N_{i+1} + \lfloor \frac{N_i'}{2} \rfloor$, from which (2) and (3) follow as in [10]. First,

$$N_{i+1}' = |\{u \in anc(v) : r_u \leq i+1 \text{ and } c_u \geq i+1\}|$$
$$\leq |\{u \in anc(v) : r_u = i+1\}| + |\{u \in anc(v) : r_u \leq i \text{ and } c_u \geq i+1\}|$$
$$= N_{i+1} + |\{u \in N_i' : c_u \geq i+1\}|.$$

Thus, it is enough to prove that $|\{u \in N_i' : c_u \geq i+1\}| \leq \lfloor \frac{N_i'}{2} \rfloor$. In other words, we need to show that at most half the nodes in N_i' have their colors strictly larger than i. Suppose this is not true. Then there are at least $\lfloor \frac{N_i'}{2} \rfloor + 1$ nodes among N_i' whose colors are incremented to a value above i during the coloring procedure. In this case, there exist two nodes, say x and y, that belong to N_i' such that $c_x \geq i+1$, $c_y \geq i+1$, and there are no nodes along the path from x to y that belong to N_i'. W.l.o.g., assume that x is an ancestor of y, and let z be the node strictly between x and y with the largest color. If $c_z < i$ or if z does not exist, then the nearest ancestor with color at least c_y is x. Since $r_y \leq i$, the coloring procedure will not increase the color of y. Otherwise, if $c_z = i$, then z belongs to N_i', which is a contradiction. Finally, if $c_z > i$, the coloring procedure will not increment the color of y above i as in the earlier case. This leads to a contradiction since $c_y \geq i+1$. □

From parts (1) and (3) of Lemma 5.1, it follows that a search that starts at a node v and ends in a node u takes $O(1 + \log(W_v/w_u))$ time. The following lemma shows that one can assign colors to all the nodes in H in linear time.

LEMMA 5.2. *Given H and the weights of the nodes, we can compute all node colors in $O(n)$ time.*

Proof. To assign the colors, keep a c^{max}-bit counter (which fits into one word); the counter is initialized to 0. We perform a preorder traversal of H, and when we have visited a node v, the counter contains a 1 in bit position i (the least significant bit is position 1, and the most significant is position c^{max}) if there is an ancestor of v (including v itself) with color i such that there is no other node with color $> i$ between v and this ancestor. Upon arriving at a node v for the first time, we first compute r_v. Taking the value of the counter at v's parent to be x, we set the lowest-order

$r_v - 1$ bits of x to 1, and add 1 to the result, giving a value x'. The counter value for v is in fact x' and is stored with v. To compute the color of v, we compute the bitwise exclusive-OR of x and x' and find the position of the most significant 1 bit in the result. The implementation of the above in constant time requires standard $O(1)$ time bitwise operations, most notably the $O(1)$ time computation of the most significant bit (MSB) of a single word [26, 27].   □

Nearest colored ancestor problem. We consider the following problem: Given a rooted ordered tree T with n nodes, each of which is assigned a color from $\{1, 2, \ldots, \sigma\}$, preprocess T to answer the following query in $O(1)$ time:

$nca(v, c)$: given a node $v \in T$ and a color c, find the lowest ancestor of v in T whose color is $\geq c$.

We will use this data structure for every tree in H; clearly, the nca operation simulates following color-c pointers, thus enabling biased search. To address our application, we consider the problem in the setting where word size w is equal to the number of colors σ. Our goal is to preprocess T in $O(n)$ time and store it in a data structure of size $O(n)$ words (i.e., $O(n\sigma)$ bits) to support in $O(1)$ time not only $nca()$ but also navigation queries, such as finding the distance between an ancestor and descendant, and choosing the ith level-ancestor of a given node.

We partition the string BP of length $2n$ that stores the balanced parenthesis sequence of the given n-node tree into blocks of size $b = \min\{\sigma, \lg n\}$. Each parenthesis in BP corresponds to a node in the tree, and every node in the tree corresponds to one open and one close parenthesis in BP. Hence every node in the tree belongs to either one or two different blocks to which its corresponding parentheses in BP belong. For each block we identify a *representative* node which is the LCA of all the nodes whose corresponding parentheses (either one or both) are in that block. Thus there are $O(n/b)$ representative nodes. Our main idea is to preprocess each block so that queries whose answers lie within the block can be answered efficiently, as summarized in the following lemma. In addition, in linear time we compute and store all the answers for all the representative nodes.

LEMMA 5.3. *Given a block containing b nodes where each node is associated with a color from the range $[1, \sigma]$, one can construct a $O(n \lg \sigma)$-bit structure in $o(b)$ time such that nca queries whose answers lie within the same block can be answered in constant time.*

Proof. Our first step is to reduce the set of colors within a block from σ to $O(b)$. (If $\sigma = b$, this step is omitted.) For each block, we obtain a sorted list of all colors that appear in that block. This can be done in linear time by sorting the pairs $\langle block\_number, color_i \rangle$, where $color_i$ is the color of the ith node (i.e., the node corresponding to the ith parenthesis) in the block, using radix sort.

Let $c_1 < c_2 < \cdots < c_k$, for some $k \leq b$, be the set of all distinct colors that appear in a given block. Define $succ(c)$ to be the smallest c_i such that $c_i \geq c$. Observe that $nca(x, c) = nca(x, succ(c))$ if the answer is within the block. For each block, we store the sorted sequence $c_1, c_2, \ldots, c_k$ of all distinct colors that appear in the block using an atomic heap [28], to support $succ()$ queries in constant time.

The range of colors in each block is now reduced to at most b. Thus, we need to answer the $nca()$ query in a block of size b, where the nodes are associated with colors in the range $[1, b]$. Using $b \lg b$ bits, we store the string consisting of the "reduced" colors of the nodes in the same order as the nodes in the block. For each color c, $1 \leq c \leq b$, we build an $o(b)$-bit auxiliary structure that enables us to answer the query $nca(x, c)$ in constant time, for any node x in the block if the answer lies within

the block.

We divide each block (of size $b = \lg n$) into subblocks of size $s = \epsilon \lg n / \lg \lg n$ for some positive constant $\epsilon < 1$. If the answer to an $nca()$ query lies in the same subblock as the query node, then we can find the answer using precomputed tables, as all the information related to a subblock (the parenthesis sequence and the "reduced" color information of the nodes) fits in $O(\lg n)$ bits—the constant factor can be made less than $1/2$ by choosing the parameter ϵ in the subblock size appropriately. If the answer to the query does not lie in the same subblock, but within the same block, then we first determine the subblock (within the block) that contains the answer. To do this efficiently, we store the following additional information for each block.

Given a reduced color c in the block and a position i within the block (corresponding to a node x), we define the *colored excess* of the position (with respect to the representative of the block) as the number of nodes with color c in the path from x to $rep(x)$. For every reduced color in the range $[1 \ldots b]$ and every subblock we compute and store the minimum and maximum colored excess values within the subblock. Using this information for all the subblocks within a block, and for any particular color, we can find the subblock containing the answer to a query with respect to that color (in constant time, using precomputed tables of negligible size). As there are b/s subblocks and b colors within each block, and the values stored for each subblock are in the range $[0 \ldots b]$, the information stored for each block is $O((b/s)b \lg b) = O(b(\lg \lg n)^2)$ bits. Thus, over all the blocks, the space used is $O(n(\lg \lg n)^2)$ bits, which is $o(n)$ words. The computation of this information for all the subblocks can be performed in $O(n)$ time as explained below.

The total size of the information we need to store for each subblock is $O(s(\lg \lg n)^2)$ $= O(b \lg \lg n)$ bits, and we need to be able to read the information corresponding to all the subblocks within a block, corresponding to any particular color, by reading a constant number of $O(\lg n)$-bit "words." For this, we divide the range of colors (i.e., the range $[1 \ldots b]$) into chunks of size $d = s/\lg \lg n$ and write down the information corresponding to all the subblocks within a block, and of all the colors within a chunk, which fits in $O(\lg n)$ bits. Thus we can read the information corresponding to all the subblocks within a block, corresponding to any particular color, by reading these $O(\lg n)$-bits. We use precomputed tables to produce the information corresponding to each subblock, and for all the colors within each chunk. Hence each subblock has to be "processed" $O(b/d)$ times (as there are b/d chunks). Thus the total time spent producing the information for all the subblocks and for all the chunks for each block is $O((b/s)(b/d)) = O(\lg \lg n)^3$. Thus the overall time spent for all the blocks is $O((n/b)(\lg \lg n)^3) = o(n)$. $\square$

For each representative node x, we will store an array A of size σ such that $A_x[c] = nca(x,c)$ for $1 \leq c \leq \sigma$. As there are $O(n/b)$ representative nodes and each entry in A_x takes $\lg n$ bits, the total space used by arrays of all the representative nodes is $O((n/b)\sigma \lg n)$ bits, which is $O(n\sigma)$. We will now describe how these arrays can be constructed with linear preprocessing time.

We first prove the following properties about the representative nodes.

LEMMA 5.4. *For each node x, at least one of these three statements is true:*
(i) *$nca(x,c)$ lies in the (first) block to which x belongs,*
(ii) *$nca(x,c) = rep(x)$, or*
(iii) *$nca(x,c) = nca(rep(x),c)$.*

Proof. The lemma follows from the following two observations:
 • Either $nca(x,c) = parent(x)$ or $nca(x,c) = nca(parent(x),c)$.

- $rep(x)$ is either the highest ancestor of x that is within the block containing x, or the lowest ancestor of x that is outside the block containing x. (This follows from the fact that any block that contains nodes x and y also contains all the nodes along the path between x and y in the tree.)   ☐

LEMMA 5.5. *Each representative node (except the root) has an ancestor, within a height of at most b from its level, which is a representative node.*

Proof. Consider the lowest $b-1$ ancestors of a representative node x. Either the highest node, y, among these which is within the same block as x, or y's parent, z, is a representative node. Note that y is the LCA of all nodes between x and y, and if the block contains a sibling of y, then z is the LCA of all nodes in the block.   ☐

The root of the tree is a representative node, and the array for it consists of all null pointers. Traverse the tree in preorder, skipping all the nonrepresentative nodes. When a representative node x is reached, we will scan its ancestors starting from x up to its lowest ancestor, y, that is also a representative. Let A_y be the array stored at node y. During this upward scan, we will generate an array B of length σ as follows.

We keep track of the largest color value c^{max} encountered at any point during the upward scan, and the first c^{max} entries of the array B are filled. In each step of the scan, if we encounter a node whose color value is at most c^{max}, we simply skip this node. On the other hand, if we encounter a node whose color value, c, is larger than c^{max}, then we set the entries $B[c^{max}+1], \ldots, B[c]$ to be pointers to the current node. We also update the value c^{max} to be the new value c. We now copy A_y to another array and overwrite the first c_{max} values of A_y with the first c^{max} values of B. The resulting array is the array A_x that will be stored at node x. Generating the array B takes $O(\lg n + b)$ time, as the length of B is $O(\lg n)$, and it is "extended" at most b times. Entries of B are written using bit operations on words (note that the word size is σ). Thus the overall running time to generate all the arrays at the representative nodes is $O((n/b)(b + \lg n)) = O(n)$.

By plugging in this data structure in place of interval-biased search trees, we get part(ii) of Theorems 1.1, 1.2, and 1.3.

6. Substring decompression. We now extend our random access solutions to efficiently support substring decompression. Note that we can always decompress a substring of length m using m random access computations. In this section we show how to do it using just two random access computations and additional $O(m)$ time. This immediately implies Theorem 1.1.

We extend the representation of $\mathcal{S}$ as follows. For each node v in $\mathcal{S}$ we add a pointer to the next descendant node on the heavy-path suffix for v whose light child is to the left of the heavy-path suffix and another pointer to the next descendant node whose light child is to the right of the heavy-path suffix. This increases the space of the data structure by only a constant factor. Furthermore, we may compute these pointers during the construction of the heavy-path decomposition of $\mathcal{S}$ without increasing the asymptotic complexity.

We decompress a substring $S[i, j]$ of length $m = j-i$ as follows. First, we compute the LCA v of the search paths for i and j by doing a top-down search for i and j in parallel. (Note that one of these search paths passes through the node v, without skipping over it, since that path follows a light child from v.) We then continue the search for i and j independently. Along each heavy path on the search for i we collect all subtrees to the right of the heavy path in a linked list using the above pointers. The concatenation of the linked list is the sequence of roots of subtrees to right of the search path from v to i. Similarly, we compute the linked list of subtrees to the left

of the search path from v to j. Finally, we decode the subtrees from the linked lists, thereby producing the string $S[i, j]$.

With our added pointers we construct the linked lists in time proportional to the length of the lists, which is $O(m)$. Decoding each subtree uses time proportional to the size of the subtree. The total sizes of the subtrees is $O(m)$, and therefore decoding also takes $O(m)$ time. Adding the time for the two random access computations for i and j, we obtain Theorem 1.1.

7. Compressed approximate string matching. We now show how to efficiently solve the compressed approximate string matching problem for grammar-compressed strings. Let P be a string of length m, and let k be an error threshold. We assume that the algorithms for the uncompressed problem produce the matches in sorted order of the text positions (as is the case for all solutions that we are aware of). Otherwise, additional time for sorting should be included in the bounds.

To find all approximate occurrences of P within S without decompressing $\mathcal{S}$ we combine our substring decompression solution from the previous section with a technique for compressed approximate string matching on LZ78 and LZW compressed strings [12].

We find the occurrences of P in S in a single bottom-up traversal of $\mathcal{S}$ using an algorithm for (uncompressed) approximate string matching as a black box. At each node v in $\mathcal{S}$ we compute the matches of P in $S(v)$. If v is a leaf, we decompress the single character string $S(v)$ in constant time and run our approximate string matching algorithm. Otherwise, suppose that v has left child v_l and right child v_r. We have that $S(v) = S(v_l) \cdot S(v_r)$. We decompress the substring S' of $S(v)$ consisting of the $\min\{|S(v_l)|, m+k\}$ last characters of $S(v_l)$ and the $\min\{|S(v_r)|, m+k\}$ first characters of $S(v_r)$ and run our approximate string matching algorithm on P and S'. The list of matches of P in $S(v)$ is computed by merging the matches of P in the three lists $S(v_l)$, $S(v_r)$, and S' (we assume here that our approximate string matching algorithm produces list of matches in sorted order). This suffices since any approximate match with at most k errors starting in $S(v_l)$ and ending in $S(v_r)$ must be contained within S'.

For each node v in $\mathcal{S}$ we decompress a substring of length $O(m+k) = O(m)$, solve an approximate string matching problem between two strings of length $O(m)$, and merge lists of matches. Since there are n nodes in $\mathcal{S}$, we do n substring decompressions and approximate string matching computations on strings of length m in total. The merging is done on disjoint matches in S and therefore takes $O(\text{occ})$ time, where occ is the total number of matches of P in S. With our substring decompression result from Theorem 1.1 and an arbitrary approximate string matching algorithm we obtain Theorem 1.2.

8. Random access to compressed trees. We now consider the problem of performing operations on "SLP-compressed" trees. The raw data is an ordered rooted tree T (of arbitrary degree) with N nodes. We assume that the nodes of T are numbered from 1 to N in preorder, and that T is represented by an SLP $\mathcal{S}$ that generates the BP sequence of T [50]. As noted in the introduction, this model captures existing tree compression methods. We illustrate this by showing that the SLP can asymptotically match a common tree compression technique, where T is compressed by sharing identical subtrees, giving a DAG with n nodes (see Figure 7).

LEMMA 8.1. *Given a rooted ordered tree T with N nodes that is compressed to a rooted DAG G with e edges and $O(e)$ nodes, the BP string of T can be represented by an SLP of size $O(e)$.*
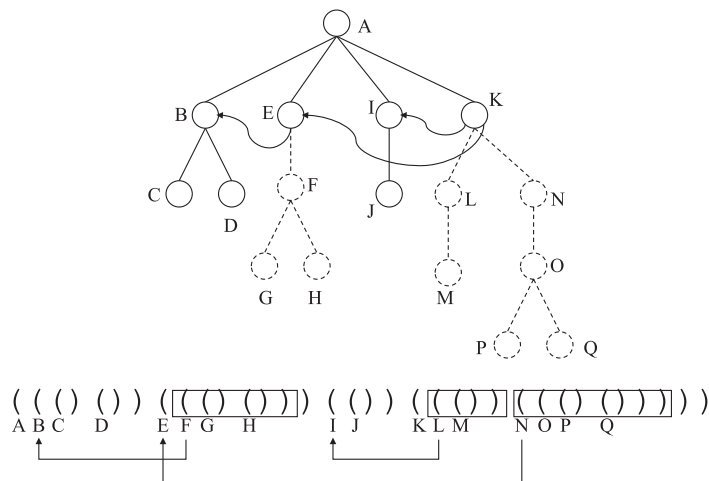
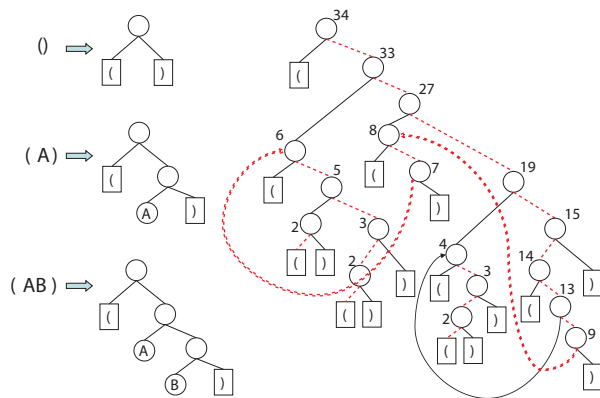Fig. 7. *A compressed tree given as a DAG and its BP representation.*

Fig. 8. *The SLP representing the BP string of the tree in Figure 7—numbers above an internal node (nonterminal) represent the lengths of strings output by that nonterminal. Heavy paths are shown by red/dashed lines—there are three in all.*

Proof. Create an SLP that generates the BP string of the tree T as follows. For each node x of the DAG G with $k \geq 0$ children, we create $k+1$ nonterminal nodes $x_0, \ldots, x_k$. The nodes x_0 and x_k have "(" and ")" as their left and right child, respectively. The node x_i $(0 \leq i \leq k-1)$ has x_{i+1} as its right child, and the node x_j $(1 \leq j \leq k)$ has the corresponding representation of the jth child of the node x of G (see Figure 8). Clearly, the size of this SLP is $O(e)$.   □

We first consider computing some functions on a (binary) string S, when the string is given as an SLP $\mathcal{S}$.

rank(S, i): Returns the number of 1's in $S[1] \ldots S[i]$.

select(S, i): Returns the position of the ith 1 in S.

excess(S, i): Returns the difference between the number of 1's and the number of 0's in $S[1] \ldots S[i]$.

We omit the first argument if it is clear from the context. In addition, we will use

excess to denote both the mathematical quantity and the operation above. If S is a BP string representing a tree T, with "(" encoded as 1 and ")" encoded as 0, then excess(i), if the position i is such that $S[i] = $ "(", is just the depth of the node represented by that opening parenthesis.

We now introduce some notation. For any binary string s, denote the number of 1's in s by weight(s), and for convenience define sum(s) = excess($s, |s|$) = $2 \cdot$ weight(s) $- |s|$ as the difference between the numbers of 1's and 0's in s. We now show the following result.

LEMMA 8.2. *For an SLP $\mathcal{S}$ of size n representing a binary string S of length N we can support the operations* rank, select, *and* excess *in $O(\log N)$ time, and $O(n\alpha_k)$ space and preprocessing time in the pointer machine model, and in linear space and preprocessing time on the RAM model.*

Proof. In what follows, we will use $\mathcal{T}$ to denote the parse tree of the given string to avoid confusion with the rooted ordered tree T that we eventually aim to represent. For any node $v \in \mathcal{T}$, abbreviate weight($S(v)$) and sum($S(v)$) as weight(v) and sum(v), respectively. If we store sum(v) and weight(v) values at each node in $\mathcal{T}$ in addition to the size values, it is straightforward to perform rank(i) and excess(i) by walking down the $\mathcal{T}$ to the ith symbol and accumulating weight/sum values from nodes to the left of the search path in $O(h)$ time, where h is the height of $\mathcal{T}$, and $O(N)$ space.

To do this in $O(\log N)$ time with $O(n)$ space, we represent the heavy-path suffix forest of $\mathcal{S}$ as in either Theorem 1.1(i) or (ii) and again traverse the DAG of $\mathcal{S}$ as if we were accessing the ith symbol of S. However, now with each node v in the DAG with heavy-path suffix $v = v_0, v_1, \ldots, v_k$, we store the total weight and total sum of the right and light children of $v_1, \ldots, v_k$ (and do the same for the left and light children). Using this information, it is easy to simulate the naive algorithm above by maintaining the invariant that after every biased search on a heavy path, upon exiting to a light node v, the accumulated values should be the same as the values accumulated by the naive algorithm at the time it reaches the node corresponding to v in $\mathcal{T}$.

It is also straightforward to perform select in $O(h)$ time on the parse tree $\mathcal{T}$ by using the weight values to guide the search to the ith 1, and accumulating size values from the nodes to the left of the search path in order to keep track of the position of this 1 in S. In order to simulate this in $O(\log N)$ time, we perform a new heavy-path decomposition on $\mathcal{S}$ using weight(v) to determine whether v is a heavy or light child. In addition, we keep with each heavy-path suffix the sums of size values to the light nodes on the right and left sides of the heavy-path suffix, and simulate the naive select algorithm using biased search on the heavy-path suffix forest as described above.   ☐

A major tool for navigating in trees represented as a BP sequence is *excess search*. Specifically, the operations to be supported are:

    fwd_search(i, δ): Given a position i and an integer δ (δ may be positive, negative, or zero), return the smallest $j > i$ such that excess(j) = excess(i) + δ and -1 if no such position exists.

    bwd_search(i, δ): As fwd_search, except that it returns the largest $j < i$.

In addition, the following operations on the BP sequence are useful in supporting a few additional navigational operations on the tree [58]:

    rmq(i, j): Return the minimum value of excess(k), where $i \leq k \leq j$.

    rmqi(i, j): Return an index k such that $i \leq k \leq j$ and excess(k) = rmq(i, j).

    RMQ(i, j): Return the maximum value of excess(k), where $i \leq k \leq j$.

    RMQi(i, j): Return an index k such that $i \leq k \leq j$ and excess(k) = RMQ(i, j).
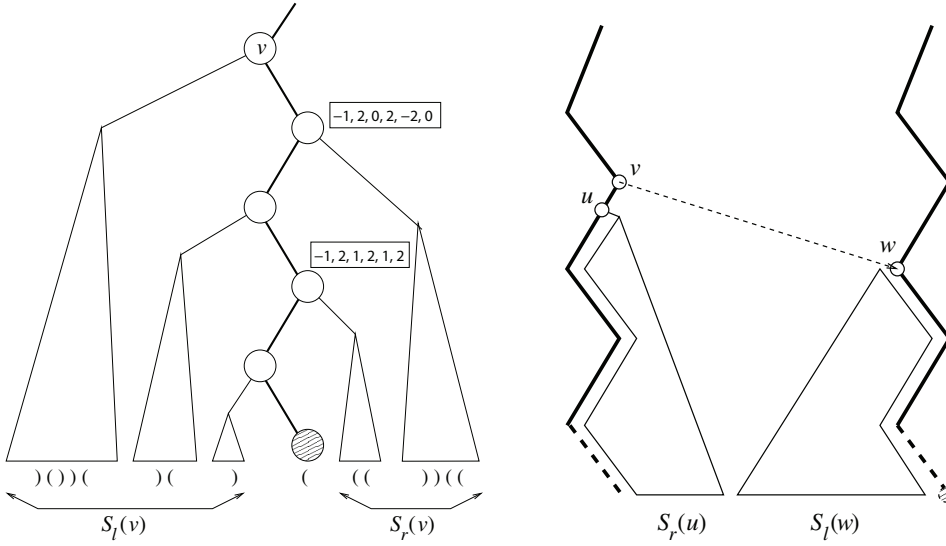
FIG. 9. *Left:* $S_l(v)$ *and* $S_r(v)$ *for a node in* $\mathcal{T}$. *Also shown with the descendants of v with right and light children are the values* $\mathrm{sum}_l, \mathrm{sum}_r, m_r, M_r, \bar{m}_r, \bar{M}_r$. *Right: Search (downward) moving from one heavy path to another.*

We now introduce some further notation. Define $M(s) = \mathrm{RMQ}(1, |s|)$ as the maximum excess value attained at any position in s. Define $m(s) = \mathrm{rmq}(1, |s|)$ analogously as the minimum excess value attained at any position in s. Note that s need not be a binary string representing a balanced parenthesis sequence, so $m(s)$ can be negative. The *excess range* of a string s is $[m(s), M(s)]$. As consecutive prefixes of s have excess values that differ by ± 1, every excess value within the excess range of s will be achieved by some prefix of s. We now show the following.

LEMMA 8.3. *For an SLP $\mathcal{S}$ of size n representing a binary string S of length N we can support the operations* fwd_search, bwd_search, rmq, rmqi, RMQ, *and* RMQi, *all in $O(\log N)$ time, and using $O(n\alpha_k)$ space and preprocessing time in the pointer machine model, and in linear space and preprocessing time on the RAM model.*

Proof. The basic idea is to simulate excess search in a manner similar to the *min-max* tree [58], with the difference that the (logical) min-max tree is built upon the parse tree itself, and also that excess search in the min-max tree when it is represented as a DAG introduces some additional challenges. Our description focuses on fwd_search(i, δ), as bwd_search(i, δ) is symmetric (however, note that fwd_search may use bwd_search and vice versa).

For any node v in the DAG of $\mathcal{S}$ with heavy path suffix $v = v_0, v_1, \ldots, v_k$, let $S_l(v)$ $(S_r(v))$ be the concatenation of the strings generated by the left and light (right and light) children of v_i, $i = 0, \ldots, k$ (see Figure 9). We store the following data with v, in addition to the data already stored for random access: $m(S_r(v))$, $M(S_r(v))$, $\mathrm{sum}(S_l(v))$, and $\mathrm{sum}(S_r(v))$, abbreviated as $m_r(v)$, $M_r(v)$, $\mathrm{sum}_l(v)$, and $\mathrm{sum}_r(v)$ (the asymmetry is because we focus on fwd_search for now). Finally, suppose that v's light child u is a right child. Then define $\bar{M}_r(v)$ as the maximum excess obtained within $S(u)$, when $S(u)$ is considered as a substring of $S_r(v)$, i.e., $\bar{M}_r(v) = M(S(u)) + \mathrm{sum}_r(w)$, where $w = v_1$ is the heavy child of v. If v's light child is a left child, we take $\bar{M}_r(v)$ as $-\infty$. Define $\bar{m}_r(v)$ analogously. Create a range maximum

query data structure [33] on each heavy path (if using the data structure of section 4), or a tree range maximum query data structure [23] (if using the biased skip tree of section 5), over the values $\bar{M}_r(v)$, and similarly create a range minimum data structure for $\bar{m}_r(v)$. These data structures do not increase the asymptotic space complexity, and they answer any range minimum/maximum queries that we require in $O(1)$ time.

The operation fwd_search(i, δ) is done in three phases. First, we search for the ith parenthesis. Next, we retrace the path taken in the search backwards (in the direction of the root of $\mathcal{T}$), looking for the node in $\mathcal{S}$ that represents the LCA in $\mathcal{T}$ of i and j, where j is the (unknown) position sought. Finally, we search downwards to j.

The first step proceeds as previously. By construction, this path passes through $O(\log N)$ light edges: we record these edges. In the second, consider first the retracing of a light edge (v, w), where v is an ancestor of w, and assume that it has been previously checked that the LCA of i and j is a proper ancestor of w. We first check that v is not the desired LCA, by checking the excess range at the right heavy child of v (if v has no right heavy child, it is anyway not the LCA). Now suppose that v is not the desired LCA and that (v', w') is the next light edge to consider. Then w' is a heavy ancestor of v, and we need to check whether the LCA lies on the heavy path $w' \to v$. This is done by performing a range maximum and minimum query on the path $w' \to v$, to find the largest value M^* of $\bar{M}_r$ and the smallest value m^* of $\bar{m}_r$ achieved on this path. If the sought excess value does not lie in the interval $[m^*, M^*]$, then the LCA does not lie on the path $w' \to v$, and we consider (v', w') as before. If the sought excess value lies within $[m^*, M^*]$, then the sought LCA lies on the path $w' \to v$ (the correctness of this argument relies on the fact that excess values change by ± 1 per position). Once we have determined that the sought LCA lies on the path $w' \to v$, we can find the LCA using binary search in $O(\log N)$ time using the range minimum/maximum queries as above. If the LCA does not lie on the path $w' \to v$, we next consider the light edge (v', w').

Once the LCA x is found, we move to x's right light child y and begin a series of biased searches along y's heavy-path suffix, essentially as in the random access case. We first check to see whether the desired excess is achieved in either $S_l(y)$ or in the nonterminal at the root of a heavy path. If not, it must be achieved in $S_r(y)$. This check can be done by looking at the excess range of the light child of y. If the desired excess is in $S_r(y)$, we need to find the node z on the heavy path that is closest to the root of the heavy-path tree that still has the desired excess in $[m_r(z), M_r(z)]$. The biased search is easily adapted to this scenario, and the desired node can be found in $O(\log(S(x)/S(z)))$ time as required.

A few details need to be taken into account. First, the excess range data that we store associated with $S_l(v)$ for a node v are in fact based on *backward* excesses: rather than calculating excesses of prefixes of $S_l(v)$, we calculate excesses on suffixes of $S_l(v)$. This is necessary so that meaningful values can be used for $\bar{m}_l(v)$ and $\bar{M}_l(v)$. Secondly, we often need to adjust the target excess values appropriately. For example, in Figure 9, if the target excess value sought in the heavy-path suffix containing v was j, the target excess value sought in $S(w)$ after following the light edge (v, w) is $j' = j + \text{sum}_r(u)$, where u is v's left child. If this target value is found not to lie in $S_l(w)$, then the target excess value to be searched for in $S_r(u)$ is $j' + \text{sum}_l(w) \pm 1$ (depending on whether the heavy-path suffix containing w ends in a terminal labelled 0 or 1).

The operations rmq(i, j) and RMQ(i, j) can be supported in a manner similar to fwd_search starting from position i, keeping track of the m^* and M^* values encoun-

tered so far during the retracing of the path, and limiting the search to within position j. The operations rmqi and RMQi can be immediately translated into fwd_search search once we find the rmq and RMQ values, respectively (to return the leftmost indices satisfying the required conditions).   ⬜

Given an SLP $\mathcal{S}$ of size n representing the BP sequence of a rooted ordered tree T with N nodes, we represent $\mathcal{S}$ using the data structures of Lemmas 8.2 and 8.3. Now we can support the navigational operations on tree T by using the translations of these operations to some combinations of the operations supported by Lemmas 8.2 and 8.3 (namely, rank, select, excess, fwd_search, bwd_search, rmq, rmqi, RMQ, and RMQi), as described in [58, section 3]. This completes the proof of Theorem 1.3.

9. Conclusions. Given a string S of length N that is generated by a grammar of size n, we have shown how to perform random access to a position in the string and to decompress an arbitrary substring of length m in time $O(\log N)$ and $O(m + \log N)$ time, respectively. We have also shown how to perform a wide variety of operations in $O(\log N)$ time on an N-node ordered tree represented as grammar of size n that generates a BP string representing the tree. The data structures take $O(n)$ space and preprocessing time on the RAM model (near-linear on the weaker pointer machine model). These are the first time complexities for these problems that do not have a linear dependency on the height of the grammar. Using our substring decompression as a black-box, we have given the first nontrivial results on approximate string matching in grammar-compressed strings. Our black-box method is still the fastest one to date.

Recently, Verbin and Yu [63] have described a family of strings of size N, generated by a grammar of size n, such that any data structure that uses $n^{O(1)}$ words of space must take $\Omega((\log N)^{1-\epsilon})$ time, for some constant $\epsilon > 0$, to support random access on strings from this family. They also give another family of strings of length N, generated by a grammar of size $n = \Omega(N^{1-\epsilon})$, for some constant $\epsilon > 0$, such that any data structure that uses $n(\log n)^{O(1)}$ words of space must take $\Omega(\log N/\log\log N)$ time to support random access on strings from this family. Both these lower bounds apply to our random access result since they are obtained on the cell probe model with word size $\Theta(\log N)$, which is stronger than the two models we use.

REFERENCES

[1] N. Alon and B. Schieber, *Optimal Preprocessing for Answering On-Line Product Queries*, Tech. Report TR-71/87, Institute of Computer Science, Tel Aviv University, Tel Aviv, Israel, 1987.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe, *Marked ancestor problems*, in Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Press, Piscataway, NJ, 1998, pp. 534–543.

[3] A. Amir, G. Benson, and M. Farach, *Let sleeping files lie: Pattern matching in Z-compressed files*, J. Comput. System Sci., 52 (1996), pp. 299–307.

[4] A. Amir, G. M. Landau, and D. Sokol, *Inplace 2D matching in compressed images*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, Philadelphia, 2003, pp. 853–862.

[5] A. Amir, M. Lewenstein, and E. Porat, *Faster algorithms for string matching with k mismatches*, J. Algorithms, 50 (2004), pp. 257–275.

[6] A. Apostolico and S. Lonardi, *Some theory and practice of greedy off-line textual substitution*, in Proceedings of the IEEE Data Compression Conference, 1998, pp. 119–128.

[7] A. Apostolico and S. Lonardi, *Compression of biological sequences by greedy off-line textual substitution*, in Proceedings of the IEEE Data Compression Conference, 2000, pp. 143–152.

[8] A. Apostolico and S. Lonardi, *Off-line compression by greedy textual substitution*, Proc. IEEE, 88 (2000), pp. 1733–1744.

[9] O. ARBELL, G. M. LANDAU, AND J. MITCHELL, *Edit distance of run-length encoded strings*, Inform. Process. Lett., 83 (2001), pp. 307–314.

[10] A. BAGCHI, A. L. BUCHSBAUM, AND M. GOODRICH, *Biased skip lists*, Algorithmica, 42 (2005), pp. 31–48.

[11] S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, *Biased search trees*, SIAM J. Comput., 14 (1985), pp. 545–568.

[12] P. BILLE, R. FAGERBERG, AND I. L. GØRTZ, *Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts*, ACM Trans. Algorithms, 6 (2010), 3.

[13] P. BUNEMAN, B. CHOI, W. FAN, R. HUTCHISON, R. MANN, AND S. VIGLAS, *Vectorizing and querying large XML repositories*, in Proceedings of the International Conference on Data Engineering (ICDE), IEEE Computer Society, Piscataway, NJ, 2005, pp. 261–272.

[14] H. BUNKE AND J. CSIRIK, *An improved algorithm for computing the edit distance of run length coded strings*, Inform. Process. Lett., 54 (1995), pp. 93–96.

[15] G. BUSATTO, M. LOHREY, AND S. MANETH, *Efficient memory representation of XML document trees*, Inform. Syst., 33 (2008), pp. 456–474.

[16] P. CÉGIELSKI, I. GUESSARIAN, Y. LIFSHITS, AND Y. MATIYASEVICH, *Window subsequence problems for compressed texts*, in Proceedings of the 1st Symposium on Computer Science in Russia (CSR), 2006, pp. 127–136.

[17] M. CHARIKAR, E. LEHMAN, D. LIU, R. PANIGRAHY, M. PRABHAKARAN, A. SAHAI, AND A. SHELAT, *The smallest grammar problem*, IEEE Trans. Inform. Theory, 51 (2005), pp. 2554–2576.

[18] B. CHAZELLE AND B. ROSENBERG, *Computing partial sums in multidimensional arrays*, in Proceedings of the 5th Annual ACM Symposium on Computational Geometry (SCG), 1989, pp. 131–139.

[19] F. CLAUDE AND G. NAVARRO, *Self-indexed text compression using straight-line programs*, in Proceedings of the 34th Conference on Mathematical Foundations of Computer Science, Lecture Notes in Comput. Sci. 5734, Springer, New York, 2009, pp. 235–246.

[20] R. COLE AND R. HARIHARAN, *Approximate string matching: A simpler faster algorithm*, SIAM J. Comput., 31 (2002), pp. 1761–1782.

[21] M. CROCHEMORE, G. M. LANDAU, AND M. ZIV-UKELSON, *A subquadratic sequence alignment algorithm for unrestricted scoring matrices*, SIAM J. Comput., 32 (2003), pp. 1654–1673.

[22] O. DELPRATT, R. RAMAN, AND N. RAHMAN, *Engineering succinct dom*, in Proceedings of the International Conference on Extending Database Technology (EDBT), ACM Internat. Conf. Proc. Ser. 261, A. Kemper, P. Valduriez, N. Mouaddib, J. Teubner, M. Bouzeghoub, V. Markl, L. Amsaleg, and I. Manolescu, eds., ACM, New York, 2008, pp. 49–60.

[23] E. D. DEMAINE, G. M. LANDAU, AND O. WEIMANN, *On Cartesian trees and range minimum queries*, in Lecture Notes in Comput. Sci. 5555, S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, and W. Thomas, eds., Springer, New York, 2009, pp. 341–353.

[24] M. FARACH AND S. MUTHUKRISHNAN, *Perfect hashing for strings: Formalization and algorithms*, in Proceedings of the 7th Symposium on Combinatorial Pattern Matching (CPM), Springer, New York, 1996, pp. 130–140.

[25] P. FERRAGINA, F. LUCCIO, G. MANZINI, AND S. MUTHUKRISHNAN, *Compressing and indexing labeled trees, with applications*, J. ACM, 57 (2009), 4.

[26] F. E. FICH, *Constant-time Operations for Words of Length w*, manuscript, 1999.

[27] M. L. FREDMAN AND D. E. WILLARD, *Surpassing the information theoretic bound with fusion trees*, J. Comput. System Sci., 47 (1993), pp. 424–436.

[28] M. L. FREDMAN AND D. E. WILLARD, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Comput. System Sci., 48 (1994), pp. 533–551.

[29] P. GAGE, *A new algorithm for data compression*, The C Users J., 12 (1994), pp. 23–38.

[30] L. GASIENIEC, R. KOLPAKOV, I. POTAPOV, AND P. SANT, *Real-time traversal in grammar-based compressed files*, in Proceedings of the Data Compression Conference, 2005, pp. 458–458.

[31] L. GASIENIEC AND I. POTAPOV, *Time/space efficient compressed pattern matching*, Fund. Inform., 56 (2003), pp. 137–154.

[32] T. HAGERUP, *Sorting and searching on the word ram*, in Proceedings of STACS, Lecture Notes in Comput. Sci. 1373, M. Morvan, C. Meinel, and D. Krob, eds., Springer, New York, 1998, pp. 366–398.

[33] D. HAREL AND R. E. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[34] D. HERMELIN, G. M. LANDAU, S. LANDAU, AND O. WEIMANN, *A unified algorithm for accelerating edit-distance via text-compression*, in Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS), Springer, New York, 2009, pp. 529–540.

[35] J. IACONO, *private communication*, NYU, New York, 2010.

[36] G. JACOBSON, *Space-efficient static trees and graphs*, in Proceedings of the 30th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Piscataway, NJ, 1989, pp. 549–554.

[37] J. JANSSON, K. SADAKANE, AND W. SUNG, *Ultra-succinct representation of ordered trees with applications*, J. Comput. System Sci., 78 (2012), pp. 619–631.

[38] J. KARKKAINEN, G. NAVARRO, AND E. UKKONEN, *Approximate string matching over Ziv-Lempel compressed text*, in Proceedings of the 11th Symposium on Combinatorial Pattern Matching (CPM), Springer, New York, 2000, pp. 195–209.

[39] J. KARKKAINEN AND E. UKKONEN, *Lempel-Ziv parsing and sublinear-size index structures for string matching*, in Proceedings of the 3rd South American Workshop on String Processing (WSP), 1996, pp. 141–155.

[40] T. KIDA, T. MATSUMOTO, Y. SHIBATA, M. TAKEDA, A. SHINOHARA, AND S. ARIKAWA, *Collage system: A unifying framework for compressed pattern matching*, Theoret. Comput. Sci., 1 (2003), pp. 253–272.

[41] J. C. KIEFFER AND E. H. YANG, *Grammar based codes: A new class of universal lossless source codes*, IEEE Trans. Inform. Theory, 46 (2000), pp. 737–754.

[42] J. C. KIEFFER, E. H. YANG, G. J. NELSON, AND P. COSMAN, *Universal lossless compression via multilevel pattern matching*, IEEE Trans. Inform. Theory, 46 (2000), pp. 1227–1245.

[43] D. E. KNUTH, *Optimum binary search trees*, Acta Inform., 1 (1971), pp. 14–25.

[44] G. M. LANDAU AND U. VISHKIN, *Fast parallel and serial approximate string matching*, J. Algorithms, 10 (1989), pp. 157–169.

[45] J. N. LARSSON AND A. MOFFAT, *Off-line dictionary-based compression*, Proc. IEEE, 88 (2000), pp. 1722–1732.

[46] Y. LIFSHITS, *Processing compressed texts: A tractability border*, in Proceedings of the 18th Symposium on Combinatorial Pattern Matching (CPM), Springer, New York, 2007, pp. 228–240.

[47] V. MAKINEN, G. NAVARRO, AND E. UKKONEN, *Approximate matching of run-length compressed strings*, in Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM), Springer, New York, 1999, pp. 1–13.

[48] U. MANBER, *A text compression scheme that allows fast searching directly in the compressed file*, in Proceedings of the 5th Symposium on Combinatorial Pattern Matching (CPM), Springer, New York, 1994, pp. 31–49.

[49] K. MEHLHORN, *Nearly optimal binary search trees*, Acta Inform., 5 (1975), pp. 287–295.

[50] J. I. MUNRO AND V. RAMAN, *Succinct representation of balanced parentheses and static trees*, SIAM J. Comput., 31 (2001), pp. 762–776.

[51] J. I. MUNRO AND S. S. RAO, *Succinct representations of functions*, in Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), Springer, New York, 2004, pp. 1006–1015.

[52] G. NAVARRO, *A guided tour to approximate string matching*, ACM Comput. Surv., 33 (2001), pp. 31–88.

[53] G. NAVARRO, T. KIDA, M. TAKEDA, A. SHINOHARA, AND S. ARIKAWA, *Faster approximate string matching over compressed text*, in Proceedings of the 11th Data Compression Conference (DCC), IEEE Press, Piscataway, NJ, 2001, pp. 459–468.

[54] G. NAVARRO AND A. O. PEREIRA, *Faster compressed suffix trees for repetitive text collections*, in Proceedings of the 13th International Symposium on Experimental Algorithms (SEA), Lecture Notes in Comput. Sci. 8504, Springer, New York, 2014, pp. 424–435.

[55] C. G. NEVILL-MANNING AND I. H. WITTEN, *Identifying hierarchical structure in sequences: A linear-time algorithm*, J. Artificial Intelligence Res., 7 (1997), pp. 67–82.

[56] M. PĂTRAŞCU, *private communication*, 2009.

[57] W. RYTTER, *Application of Lempel-Ziv factorization to the approximation of grammar-based compression*, Theoret. Comput. Sci., 302 (2003), pp. 211–222.

[58] K. SADAKANE AND G. NAVARRO, *Fully-functional succinct trees*, in Proceedings of the Twenty-first ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, Philadelphia, 2010, pp. 134–149.

[59] P. SELLERS, *The theory and computation of evolutionary distances: Pattern recognition*, J. Algorithms, 1 (1980), pp. 359–373.

[60] Y. SHIBATA, T. KIDA, S. FUKAMACHI, M. TAKEDA, A. SHINOHARA, T. SHINOHARA, AND S. ARIKAWA, *Byte Pair Encoding: A Text Compression Scheme That Accelerates Pattern Matching*, Technical Report DOI-TR-161, Department of Informatics, Kyushu University, Fukuoka, Japan, 1999.

[61] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, *Speeding up pattern matching by text compression*, in Proceedings of the 4th Italian Conference Algorithms and Complexity (CIAC), Springer, New York, 2000, pp. 306–315.

[62] R. E. Tarjan, *Data Structures and Network Algorithms*, CBMS-NSF Regional Conf. Ser. in Appl. Math. 44, SIAM, Philadelphia, 1983.

[63] E. Verbin and W. Yu, *Data structure lower bounds on random access to grammar-compressed strings*, in Proceedings of CPM, Lecture Notes in Comput. Sci. 7922, J. Fischer and P. Sanders, eds., Springer, New York, 2013, pp. 247–258.

[64] T. A. Welch, *A technique for high-performance data compression*, Computer, 17 (1984), pp. 8–19.

[65] E. H. Yang and J. C. Kieffer, *Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform—Part one: Without context models*, IEEE Trans. Inform. Theory, 46 (2000), pp. 755–754.

[66] J. Ziv and A. Lempel, *On the complexity of finite sequences*, IEEE Trans. Inform. Theory, 22 (1976), pp. 75–81.

[67] J. Ziv and A. Lempel, *A universal algorithm for sequential data compression*, IEEE Trans. Inform. Theory, 23 (1977), pp. 337–343.