Size-Constrained Weighted Ancestors with Applications

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Yakov Nekrich
Michigan Technological University, Houghton, MI, US

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

Abstract

The *weighted ancestor* problem on a rooted node-weighted tree T is a generalization of the classic *predecessor* problem: construct a data structure for a set of integers that supports fast predecessor queries. Both problems are known to require $\Omega(\log \log n)$ time for queries provided $\mathcal{O}(n \text{ poly} \log n)$ space is available, where n is the input size. The weighted ancestor problem has attracted a lot of attention by the combinatorial pattern matching community due to its direct application to suffix trees. In this formulation of the problem, the nodes are weighted by string depth. This research has culminated in a data structure for weighted ancestors in suffix trees with $\mathcal{O}(1)$ query time and an $\mathcal{O}(n)$-time construction algorithm [Belazzougui et al., CPM 2021].

In this paper, we consider a different version of the weighted ancestor problem, where the nodes are weighted by any function weight that maps each node of T to a positive integer, such that $\mathsf{weight}(u) \leq \mathsf{size}(u)$ for any node u and $\mathsf{weight}(u_1) \leq \mathsf{weight}(u_2)$ if node u_1 is a descendant of node u_2, where $\mathsf{size}(u)$ is the number of nodes in the subtree rooted at u. In the *size-constrained weighted ancestor* (SWA) problem, for any node u of T and any integer k, we are asked to return the lowest ancestor w of u with weight at least k. We show that for *any rooted tree* with n nodes, we can locate node w in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing. In particular, this implies a data structure for the SWA problem in suffix trees with $\mathcal{O}(1)$ query time and $\mathcal{O}(n)$-time preprocessing, when the nodes are weighted by weight. We also show several string-processing applications of this result.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases weighted ancestors, string indexing, data structures

Digital Object Identifier 10.4230/LIPIcs.SWAT.2024.14

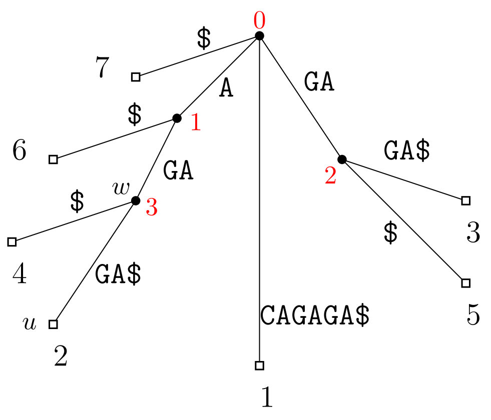
Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2311.15777`

Funding *Philip Bille*: Supported by the Independent Research Fund Denmark (DFF-9131-00069B).
Yakov Nekrich: Supported by the National Science Foundation under NSF grant 2203278.
Solon P. Pissis: Supported by the PANGAIA (No 872539) and ALPACA (No 956229) projects.

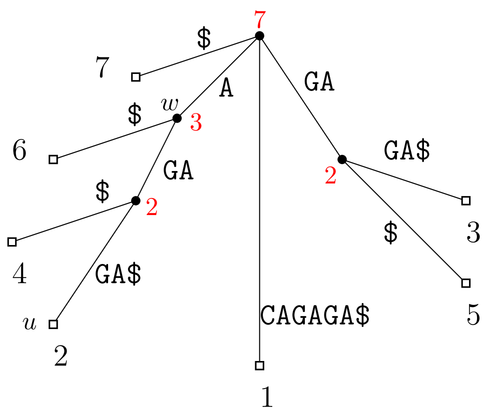
1 Introduction

In the classic *predecessor* problem [27, 16, 29, 24, 23], we are given a set S of keys from a universe U with a total order. The goal is to preprocess set S into a compact data structure supporting the following on-line queries: for any element $q \in U$, return the maximum $p \in S$ such that $p \leq q$; p is called the *predecessor* of q.

The *weighted ancestor* problem, introduced by Farach and Muthukrishnan in [15], is a natural generalization of the predecessor problem on rooted node-weighted trees. In particular, given a rooted tree T, whose nodes are weighted by positive integers and such that these weights decrease when ascending from any node to the root, the goal is to preprocess



(a) The internal nodes are weighted by *string depth* (in red). Asking a weighted ancestor query for $i = 2$ (node u) and $k = 2$ will take us to node w. Indeed, (w, k) is the locus of substring `AG` in the suffix tree of X.



(b) The internal nodes are weighted by *frequency* (in red). Asking a weighted ancestor query for $i = 2$, $j = 7$ (node u) and $k = 3$ will take us to node w. Indeed, `A` is the longest prefix of `AGAGA$` that occurs at least 3 times in X.

Figure 1 Weighted ancestor queries on the suffix tree of string $X =$ `CAGAGA$`. The leaf nodes in both trees are labeled by the starting position of the suffix of X they represent.

tree T into a compact data structure supporting the following on-line queries: for any given node u and any integer $k > 0$, return the farthest ancestor of u whose weight is at least k. Both the predecessor and the weighted ancestor problems require $\Omega(\log \log n)$ time for queries provided $\mathcal{O}(n \operatorname{poly} \log n)$ space is available, where n is the input size of the problem [17].

The weighted ancestor problem has attracted a lot of attention in the combinatorial pattern matching community [15, 4, 22, 21, 17, 8, 6] due to its direct application to suffix trees [28]. The *suffix tree* of a string X is the compacted trie of the set of suffixes of X; see Figure 1a. In this formulation of the problem, a node u is weighted by *string depth*: the length of the string spelled from the root of the suffix tree to u; and a weighted ancestor query for two integers i and $k > 0$ returns the locus of substring $X[i \mathinner{.\,.} i + k - 1]$ in the suffix tree of X. We refer the reader to [17] for several applications. This research has culminated in a data structure for weighted ancestors in suffix trees, given by Belazzougui, Kosolobov, Puglisi, and Raman [8], supporting $\mathcal{O}(1)$-time queries after an $\mathcal{O}(n)$-time preprocessing.

However there are other tree weighting schemes that are of interest to string processing. For example, each suffix tree node can be weighted by the number of its leaf descendants; see Figure 1b. Thus the weight of a node u is equal to the frequency of the substring represented by the root-to-u path. If we use this weighting function, then the following basic string problem can be translated into a weighted ancestor query: *Given a substring $I = X[i \mathinner{.\,.} j]$ of string X and an integer $k > 0$, find the longest prefix of I that occurs at least k times in X.*

Unfortunately, the existing data structures for the weighted ancestor problem *on suffix trees* [17, 8] depend strongly on the fact that the suffix tree nodes are weighted by string depth. They thus *cannot be applied* to solve the aforementioned basic string problem.

Motivated by this fact, we introduce a different version of the weighted ancestor problem on *general rooted trees*. Let T be a rooted tree on a set V of n nodes. By $\mathsf{size}(u)$, we denote the number of nodes in the subtree rooted at a node $u \in V$. Let $\mathsf{weight} : V \to \mathbb{N}$ denote any function that maps each node of T to a positive integer, such that $\mathsf{weight}(u) \leq \mathsf{size}(u)$ for any node $u \in V$ and $\mathsf{weight}(u_1) \leq \mathsf{weight}(u_2)$ if node $u_1 \in V$ is a descendant of node $u_2 \in V$. The latter is also known as the *max-heap property*: the weight of each node is less than or equal to the weight of its parent, with the maximum-weight element at the root. We will say that a function $\mathsf{weight} : V \to \mathbb{N}$ satisfying both properties is a *size-constrained max-heap* weight function. For any node $u \in V$ and any integer $k > 0$, a *size-constrained weighted ancestor query*, denoted by $\mathsf{SWA}(u, k) = w$, asks for the lowest ancestor $w \in V$ of u with weight at least k. The *size-constrained weighted ancestor* (SWA) problem, formalized next, is to preprocess T into a compact data structure supporting fast SWA queries:

Size-Constrained Weighted Ancestor (SWA)
Preprocess: A rooted tree T on a set V of n nodes weighted by a size-constrained max-heap function $\mathsf{weight} : V \to \mathbb{N}$.
Query: Given a node $u \in V$ and an integer $k > 0$, return the lowest ancestor w of u with $\mathsf{weight}(w) \geq k$.

We assume throughout the standard word RAM model of computation with word size $\Theta(\log n)$; basic arithmetic and bit-wise operations on $\mathcal{O}(\log n)$-bit integers take $\mathcal{O}(1)$ time. Note that, since function weight must satisfy the max-heap property, one can employ the existing data structures for the weighted ancestor problem *on general rooted trees* [15, 4], to answer SWA queries in $\mathcal{O}(\log \log n)$ time after $\mathcal{O}(n)$-time preprocessing (see also [25]). Our main result in this paper can be formalized as follows (see Section 3 and Section 4).

▶ **Theorem 1.** *For any rooted tree with n nodes weighted by a size-constrained max-heap function* weight*, there exists an $\mathcal{O}(n)$-space data structure answering* SWA *queries in $\mathcal{O}(1)$ time. The preprocessing algorithm runs in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.*

As a preliminary step, we design an $\mathcal{O}(n \log n)$-space solution using an involved combination of *rank-select* data structures [7], *fusion trees* [16], and *heavy-path decompositions* [26]. We then design a novel application of *ART decomposition* [2] to arrive to Theorem 1.

Applications. Notably, Theorem 1 presents a data structure for the SWA problem in suffix trees with $\mathcal{O}(1)$ query time and $\mathcal{O}(n)$-time preprocessing, when the nodes are weighted by a size-constrained max-heap weight function weight. We show several string-processing applications of this result since $\mathsf{weight}(u)$ can be defined as the number of leaf nodes in the subtree rooted at u. Let us first provide some intuition on the applicability of Theorem 1.

Consider a relatively long query submitted to a search-engine text database. If the database returns no (or not sufficiently many) results, one usually tries to *repeatedly* truncate some prefix and/or some suffix of the original query until they obtain sufficiently many results. Our Theorem 1 can be applied to solve this problem directly in optimal time.

In particular, Theorem 1 yields *optimal data structures*, with respect to preprocessing and query times, for the following basic string-processing problems (see Section 5):

1. Preprocess a string X into a linear-space data structure supporting the following on-line queries: for any i, j, f return the longest prefix of $X[i\mathinner{..}j]$ occurring at least f times in X.
2. Preprocess a dictionary $\mathcal{D}$ of documents into a linear-space data structure supporting the following on-line queries: for any string P and any integer f, return a longest substring of P occurring in at least f documents of $\mathcal{D}$.
3. Preprocess a string X into a linear-space data structure supporting the following on-line queries: for any string P and any integer f, return a longest substring of P occurring at least f times in X.

Theorem 1 also directly improves on the data structure presented by Pissis et al. [25] for computing the *frequency-constrained substring complexity* of a given string (see Section 5).

2 Preliminaries

For any bit string B of length m and any $\alpha \in \{0,1\}$, the classic rank and select queries are defined as follows:

- rank_α: for any given $i \in [1,m]$, it returns the number of ones (or zeros) in $B[1\,..\,i]$; more formally, $\mathsf{rank}_\alpha(B,i) = |\{j \in [1,i] : B[j] = \alpha\}|$.
- select_α: for any given rank i, it returns the leftmost position where the bit vector contains a one (or zero) with rank i; more formally, $\mathsf{select}_\alpha(B,i) = \min\{j \in [1,m] : \mathsf{rank}_\alpha(B,j) = i\}$.

The following result is known.

▶ **Lemma 2** (Rank and Select [7]). *Let B be a bit string of length $m \le n$ stored in $\mathcal{O}(1 + m/\log n)$ words. We can preprocess B in $\mathcal{O}(1 + m/\log n)$ time into a data structure of $m + o(m)$ bits supporting* rank *and* select *queries in $\mathcal{O}(1)$ time.*

Bit strings can also be used as a representation of monotonic integer sequences supporting predecessor queries; see [5], for example. Assume we have a set S of m keys from a universe U with a total order. In the *predecessor* problem, we are given a query element $q \in U$, and we are to find the maximum $p \in S$ such that $p \le q$; we denote this query by $\mathsf{predecessor}(q) = p$. The following result is known for a special case of the predecessor problem.

▶ **Lemma 3** (Fusion Tree [16]). *We can preprocess a set of $m = \log^{\mathcal{O}(1)} n$ integers in $\mathcal{O}(m)$ time and space to support* predecessor *queries in $\mathcal{O}(1)$ time.*

3 Constant-time Queries using $\mathcal{O}(n \log n)$ Space

We first show how to solve the SWA problem in $\mathcal{O}(1)$ time using $\mathcal{O}(n \log n)$ space. This solution forms the basis for our linear-time and linear-space solution in Section 4.

3.1 Heavy-path Decomposition

Let T be a rooted tree with n nodes. We compute the *heavy-path decomposition* of T in $\mathcal{O}(n)$ time [26]. Recall that, for any node u in T, we define $\mathsf{size}(u)$ to be number of nodes in the subtree of T rooted at u. We call an edge (u,v) of T *heavy* if $\mathsf{size}(v)$ is maximal among every edge originating from u (breaking ties arbitrarily). All other edges are called *light*. We call a node that is reached from its parent through a heavy edge *heavy*; otherwise, the node is called *light*. The heavy path of T is the path that starts at the root of T and at each node on the path descends to the heavy child as defined above. The *heavy-path decomposition* of T is then defined recursively: it is a union of the heavy path of T and the heavy-path decompositions of the off-path subtrees of the heavy path. A well-known property of this decomposition is that every root-to-node path in T passes through at most $\log n$ light edges. In particular, the following lemma is implied.

▶ **Lemma 4** (Heavy-path Decomposition [26]). *Let T be a rooted tree with n nodes. Any root-to-leaf path in T consists of at most $\log n + \mathcal{O}(1)$ heavy paths.*

3.2 Data Structure

We construct a heavy-path decomposition of T. Consider a heavy path $H = v_1 \dots v_\ell$. We construct a bit string $B(H)$ that represents the differences between node weights using unary coding. Suppose that nodes $v_1 \dots v_\ell$ of H are listed in decreasing order of their depth and let $\delta(v_i) = \mathsf{weight}(v_i) - \mathsf{weight}(v_{i-1})$, for all $i > 1$. We define $B(H)$ as follows:

$$B(H) = \mathsf{enc}(\mathsf{weight}(v_1)) \cdot \mathsf{enc}(\delta(v_2)) \cdot \dots \cdot \mathsf{enc}(\delta(v_i)) \cdot \dots \cdot \mathsf{enc}(\delta(v_\ell)),$$

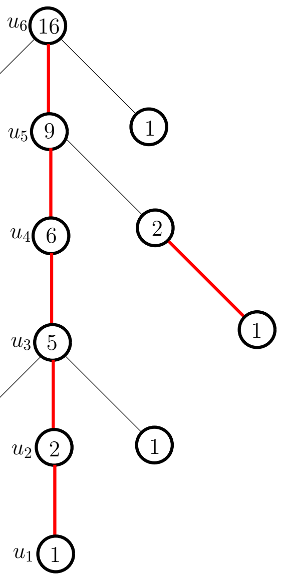
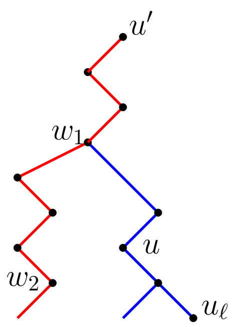


Figure 2 A rooted tree T with $n = 16$ nodes. Each node u of T is weighted by $\mathsf{weight}(u) = \mathsf{size}(u)$. For example, $\mathsf{weight}(u_5) = \mathsf{size}(u_5) = 9$, because there are 9 nodes in the subtree rooted at u_5, and $\mathsf{SWA}(u_2, 7) = u_5$ because the lowest ancestor of u_2 with weight at least 7 is node u_5. A heavy-path decomposition of T is also depicted: the heavy edges are the red edges. For example, the heavy path of the whole T is $u_1u_2 \ldots u_6$.

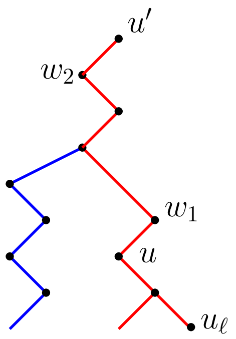
where $\mathsf{enc}(i)$ denotes the unary code of i; i.e., $\mathsf{enc}(i)$ consists of i 1's followed by a single 0. The important property of our encoding is that the total number of 0-bits in $B(H)$ is ℓ and the total number of 1-bits is $\mathsf{weight}(v_\ell)$.

▶ **Example 5.** Let $H = u_1u_2 \ldots u_6$ be the heavy path of T from Figure 2. We have $\ell = 6$ and $\mathsf{weight}(u_1) = 1$, $\mathsf{weight}(u_2) = 2$, $\mathsf{weight}(u_3) = 5$, $\mathsf{weight}(u_4) = 6$, $\mathsf{weight}(u_5) = 9$, $\mathsf{weight}(u_6) = 16$. We have $B(H) = \texttt{1010111010111011111110}$. For instance, the second 1 denotes $\delta(u_2) = \mathsf{weight}(u_2) - \mathsf{weight}(u_1) = 1$. The leftmost occurrence of 111 denotes $\delta(u_3) = \mathsf{weight}(u_3) - \mathsf{weight}(u_2) = 3$ 1's.

For any heavy path H, we can construct $B(H)$ in $\mathcal{O}(\ell)$ time using standard word RAM bit manipulations to construct the unary codes and concatenate the underlying bit strings. By Lemma 4, every leaf node of T has $\mathcal{O}(\log n)$ ancestors v_t, such that v_t is the topmost node of some heavy path H. Since any node in T is counted in the weight of $\mathcal{O}(\log n)$ topmost nodes, the total weight of all topmost nodes, summed over all heavy paths H, is $\mathcal{O}(n \log n)$. Thus, the total length of all bit strings $B(H)$ is $\mathcal{O}(n \log n)$ and we can construct them all in $\mathcal{O}(n)$ time since the total length of the heavy paths is $\mathcal{O}(n)$. We store each such bit string according to Lemma 2 to support $\mathcal{O}(1)$-time rank and select queries using $\mathcal{O}(n)$ preprocessing time and words of space. Furthermore, for each leaf node v in T we store the weights of the top nodes of each heavy path on the path from the root to v. By Lemma 4, there are $\mathcal{O}(\log n)$ such top nodes for each leaf. For every leaf node we store the weights of its top node ancestors in a *fusion tree* data structure according to Lemma 3. The total space used by all such fusion trees is $\mathcal{O}(n \log n)$ words and the preprocessing time is $\mathcal{O}(n \log n)$. Finally, we construct a *lowest common ancestor* (LCA) data structure over T. Such a data structure answers LCA queries in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space preprocessing [9].



(a) Case 1: Only w_1 is an ancestor of u. The heavy path H_w is shown in red. The $(f+1)$th node w_2 on H_w is below w_1. The node w_1 is the $(f+g)$th node on H_w for some $g > 1$, and so w_1 is the answer.



(b) Case 2: Both w_1 and w_2 are ancestors of u. The heavy path H_w is shown in red. The $(f+1)$th node w_2 on H_w is above w_1, and so w_2 is the answer.

Figure 3 The two cases of the querying algorithm.

3.3 Queries

Suppose we are given a node u and an integer k as an $\mathsf{SWA}(u,k)$ query. We are looking for the lowest ancestor w of u with weight at least k. If the weight of u is at least k, we return u. Otherwise we proceed as follows. First, we locate the heavy path H_w that contains node w: we find an arbitrary leaf descendant u_ℓ of u; then, using the fusion tree of u_ℓ, we find the lowest ancestor u' of u_ℓ with weight at least k, such that u' is a top node. H_w is the heavy path, such that u' is its top node. When we find H_w, we answer a query $f = \mathsf{rank}_0(B(H_w), j)$ for $j = \mathsf{select}_1(B(H_w), k)$ using Lemma 2 in $\mathcal{O}(1)$ time. Let w_1 denote the lowest ancestor of u on the heavy path H_w (see Figure 3). If u is on H_w (Figure 3b), then w_1 is simply the parent of u. Otherwise (Figure 3a), w_1 can be found as the lowest common ancestor of the lowest node on H_w and node u. In the latter case, w_1 can be found using an LCA query that takes $\mathcal{O}(1)$ time. Let w_2 denote the $(f+1)$th node on H_w. The node w is the highest node among w_1 and w_2. The query time is $\mathcal{O}(1)$ by Lemma 3 for finding H_w and by Lemma 2 for finding f. Example 6 shows how we use $B(H_w)$ to find f and thus the $(f+1)$th node.

▶ **Example 6.** Let $B(H) = \mathtt{10101110101110111111110}$ from Example 5, u_2 from Figure 2, and $k = 7$. Then $j = \mathsf{select}_1(B(H), 7) = 11$ and $f = \mathsf{rank}_0(B(H), 11) = 4$. The output node is u_5, the $(f+1)$th node on H. Indeed, $\mathsf{weight}(u_5) = 9 \geq k = 7$ and $\mathsf{weight}(u_4) = 6 < k = 7$.

In summary, we have shown the following result, which we will improve in the next section.

▶ **Lemma 7.** *For any rooted tree with n nodes weighted by a size-constrained max-heap function* weight*, there exists an $\mathcal{O}(n \log n)$-space data structure answering* SWA *queries in $\mathcal{O}(1)$ time. The preprocessing algorithm runs in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n \log n)$ space.*

4 Constant-time Queries using $\mathcal{O}(n)$ Space

We now improve the above solution to the SWA problem (Lemma 7) to linear-time and linear-space preprocessing. We will reuse the previous section's linear-time heavy-path decomposition and the corresponding bit string encoding. The key challenge is identifying the top nodes of heavy paths in $\mathcal{O}(1)$ time using linear space.

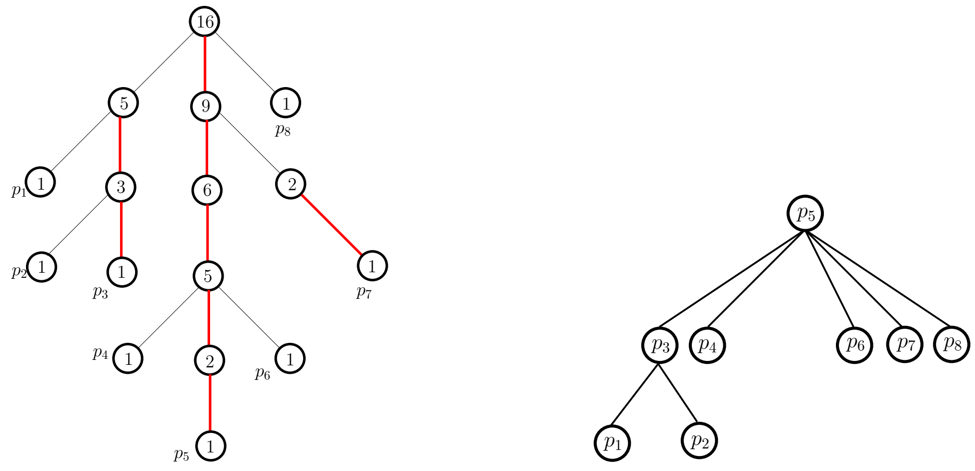
4.1 ART Decomposition

The ART decomposition, proposed by Alstrup, Husfeldt, and Rauhe [2], partitions a rooted tree into a *top tree* and several *bottom trees* with respect to an input parameter χ. Each node v of minimal depth, with no more than χ leaf nodes below it, is the root of a bottom tree consisting of v and all its descendants. The top tree consists of all nodes that are not in any bottom tree. The ART decomposition satisfies the following important property:

▶ **Lemma 8** (ART Decomposition [2]). *Let T be a rooted tree with ℓ leaf nodes. Further let χ be a positive integer. The ART decomposition of T with parameter χ produces a top tree with at most $\mathcal{O}(\ell/\chi)$ leaves. Such a decomposition of T can be computed in linear time.*

4.2 Data Structure

Recall that T consists of n nodes. As discussed in Section 3.2, we compute the heavy-path decomposition of T, construct bit strings for each heavy path, and preprocess the bit strings to support rank and select queries in $\mathcal{O}(1)$ time. This takes $\mathcal{O}(n)$ preprocessing time and space, allowing us to answer queries on a heavy path in $\mathcal{O}(1)$ time. Thus what remains is a linear-space and $\mathcal{O}(1)$-time solution to locate the top nodes of heavy paths.



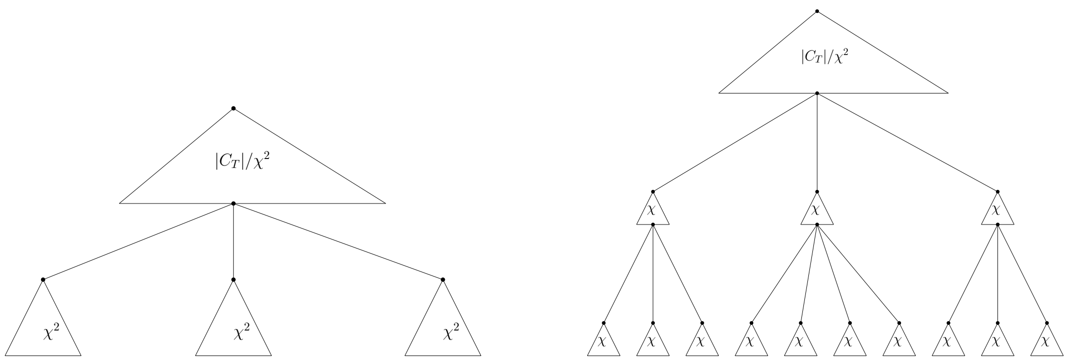
(a) The tree T from Figure 2. We write the heavy path id p_i at the end of the ith heavy path.

(b) The contracted tree C_T.

Figure 4 The contraction process of the tree T from Figure 2.

First, we construct the *contracted tree* C_T of T obtained by contracting all edges of heavy paths in T. In particular, this leaves all the light edges from T in C_T and removes all the heavy edges from T (see Figure 4). We then apply the ART decomposition on C_T (see Figure 5a) with parameter χ^2, where $\chi = \epsilon \frac{\log n}{\log \log n}$ and ϵ is a positive constant. We apply the ART decomposition again with parameter χ (see Figure 5b) on each resulting bottom tree. The resulting partition of C_T contains three levels of trees that we call the *top tree*, the *middle trees*, and the *bottom trees*. Since the heavy-path decomposition of T can be computed in $\mathcal{O}(n)$ time, contracting T takes $\mathcal{O}(n)$ time by processing the heavy-path decomposition of T. By Lemma 8, the ART decompositions of T cost $\mathcal{O}(n)$ total time.

Let us first consider the top tree. As in Section 3.2, we store a fusion tree for each leaf node in the top tree. By Lemma 8, the top tree has $\mathcal{O}(\frac{|C_T|}{\chi^2})$ leaves and hence, by Lemmas 3 and 4, this uses $\mathcal{O}(\frac{|C_T|}{\chi^2} \cdot \log n) = \mathcal{O}(\frac{n(\log \log n)^2}{\log n}) = o(n)$ space and preprocessing time.



(a) First application of ART decomposition on C_T.

(b) Application of ART decomposition on the bottom trees of the tree in Figure 5a.

Figure 5 Application of ART decompositions on C_T.

For the middle or bottom trees, we tabulate the answers to all possible queries in a global table. The index in the table is given by a tree encoding and the node u along with integer k for the SWA query. The corresponding value in the table is the output node of the $\mathsf{SWA}(u, k)$ query. We encode the input to a query as follows. We represent each middle and bottom tree compactly as a bit string encoding the tree structure and the weights of all nodes. Since each internal node in C_T is branching, the number of nodes in a middle or bottom tree is bounded by $\mathcal{O}(\chi)$. Thus, we can encode the tree structure using $\mathcal{O}(\chi)$ bits. The weight of a node in a middle or bottom tree is bounded by $\mathcal{O}(\chi^2)$ or $\mathcal{O}(\chi)$, respectively, and can thus be encoded in $\mathcal{O}(\log \chi)$ bits. Hence, we can encode the tree structure and all weights using $\mathcal{O}(\chi \log \chi)$ bits. We encode the query node u using $\mathcal{O}(\log \chi)$ bits. Since the maximum weight is $\mathcal{O}(\chi^2)$ we can also encode the query integer k using $\mathcal{O}(\log \chi)$ bits. Hence, the full encoding uses $\mathcal{O}(\chi \log \chi) + \mathcal{O}(\log \chi) + \mathcal{O}(\log \chi) = \mathcal{O}(\chi \log \chi)$ bits. To encode the output node stored in the global table we use $\mathcal{O}(\log \chi)$ bits. Thus, the table uses $2^{\mathcal{O}(\chi \log \chi)} \log \chi = 2^{\mathcal{O}(\epsilon \log n)} = o(n)$ bits for a sufficiently small constant $\epsilon > 0$. The table can be constructed in $o(n)$ time.

4.3 Queries

Suppose we are given a node u and an integer k as an $\mathsf{SWA}(u, k)$ query. Let u_t denote the top node on the heavy path of u in T and let u_H denote the corresponding node in the contracted tree C_T. We find the lowest ancestor w_H of u_H with weight at least k in C_T. If u_H is in the top tree we find w_H as described in Section 3.2. If u_H is in a middle or bottom tree, we use the global table to find w_H. If the result is not in the middle or bottom tree (the weight of the top node in such a tree is smaller than k), we move up a level and query the middle or top tree, respectively. Each of these at most three queries takes $\mathcal{O}(1)$ time. Thus w_H is found in $\mathcal{O}(1)$ time. Suppose that w_H corresponds to a node w' in the initial tree and let H' denote the heavy path such that w' is its top node. As explained in Section 3.2, we can find the lowest ancestor of u with weight at least k on H' in $\mathcal{O}(1)$ time using rank and select queries on $B(H')$. In total the $\mathsf{SWA}(u, k)$ query takes $\mathcal{O}(1)$ time.

In summary, we have obtained the following result.

▶ **Theorem 1.** *For any rooted tree with n nodes weighted by a size-constrained max-heap function* weight*, there exists an $\mathcal{O}(n)$-space data structure answering* SWA *queries in $\mathcal{O}(1)$ time. The preprocessing algorithm runs in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.*

5 String-processing Applications

In this section, we show several applications of Theorem 1 on suffix trees. Recall that the number of leaf nodes in the subtree rooted at node u in a suffix tree is the number of occurrences (i.e., the frequency) of the substring represented by the root-to-u path.

5.1 Internal Longest Frequent Prefix

Internal pattern matching is an active topic [20, 3, 12, 11, 13, 1, 5] in the combinatorial pattern matching community. We introduce the following basic string problem. The *internal longest frequent prefix* problem asks to preprocess a string X of length n over an integer alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$ into a compact data structure supporting the following on-line queries:

- $\mathsf{ILFP}_X(i, j, f)$: return the longest prefix of $X[i\,..\,j]$ occurring at least f times in X.

Our solution to this problem will form the basic tool for solving the problems in Sections 5.2 and 5.3. We first construct the suffix tree T of X in $\mathcal{O}(n)$ time [14], and preprocess it in $\mathcal{O}(n)$ time for classic weighted ancestor queries [8] as well as for SWA queries using Theorem 1. For SWA queries, as $\mathsf{weight}(u)$, we use the number of leaf nodes in the subtree rooted at node u in T. Such an assignment satisfies the requested properties of $\mathsf{weight}(\cdot)$ and can be done in linear time using a standard DFS traversal on T. Any $\mathsf{ILFP}_X(i, j, f)$ query can be answered by first finding the locus $(u, j - i + 1)$ of $X[i\,..\,j]$ in T in $\mathcal{O}(1)$ time using a classic weighted ancestor query on T, and, then, answering $\mathsf{SWA}(u, f)$ in T in $\mathcal{O}(1)$ time using Theorem 1. We obtain the following result.

▶ **Theorem 9.** *For any string X of length n over alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$, there exists an $\mathcal{O}(n)$-space data structure that answers ILFP_X queries in $\mathcal{O}(1)$ time. The preprocessing algorithm runs in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.*

5.2 Longest Frequent Substring

The *longest frequent substring* problem is the following: preprocess a dictionary $\mathcal{D}$ of d strings (documents) of total length n over an integer alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$ into a compact data structure supporting the following on-line queries:

- $\mathsf{LFS}_{\mathcal{D}}(P, f)$: return a longest substring of P that occurs in at least f documents of $\mathcal{D}$.

This longest substring of P represents a most *relevant* part of the query with respect to $\mathcal{D}$. The length of $\mathsf{LFS}_{\mathcal{D}}(P, f)$ can also be used as a *measure of similarity* between P and the strings in $\mathcal{D}$, for some f chosen appropriately based on the underlying application.

We start by constructing the generalized suffix tree T of $\mathcal{D}$ in $\mathcal{O}(n)$ time [14] and preprocess it in $\mathcal{O}(n)$ time for SWA queries using Theorem 1. For SWA queries, $\mathsf{weight}(u)$ is equal to the number of dictionary strings having at least one leaf node in the subtree rooted at node u in T. This assignment satisfies the requested properties of $\mathsf{weight}(\cdot)$ and can be done in linear time [19]. Let us denote by (v_i, ℓ_i) the locus in T of the longest prefix of $P[i\,..\,|P|]$ that occurs in any string in $\mathcal{D}$. In fact, we can compute (v_i, ℓ_i), for all $i \in [1, |P|]$, in $\mathcal{O}(|P|)$ time using the *matching statistics* algorithm of P over T [10, 18]. For each locus (v_i, ℓ_i), we trigger a $\mathsf{SWA}(v_i, f)$ query using Theorem 1 (this is essentially an instance of the *internal longest frequent prefix* problem). In total this takes $\mathcal{O}(|P|)$ time. We obtain the following result.

▶ **Theorem 10.** *For any dictionary $\mathcal{D}$ of total length n over alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$, there exists an $\mathcal{O}(n)$-space data structure that answers $\mathsf{LFS}_{\mathcal{D}}(P, f)$ queries in $\mathcal{O}(|P|)$ time. The preprocessing algorithm runs in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.*

An analogous result can be achieved for the following version of the longest frequent substring problem: preprocess a string X of length n over an integer alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$ into a compact data structure supporting the following on-line queries:

- $\mathsf{LFS}_X(P, f)$: return a longest substring of P that occurs at least f times in X.

In particular, instead of a generalized suffix tree, we now construct the suffix tree T of X and follow the same querying algorithm as above. For SWA queries, $\mathsf{weight}(u)$ is equal to the number of leaf nodes in the subtree rooted at node u in T. Such an assignment satisfies the requested properties of $\mathsf{weight}(\cdot)$ and can be done in linear time using a standard DFS traversal on T. We obtain the following result.

▶ **Theorem 11.** *For any string X of length n over alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$, there exists an $\mathcal{O}(n)$-space data structure that answers $\mathsf{LFS}_X(P, f)$ queries in $\mathcal{O}(|P|)$ time. The preprocessing algorithm runs in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.*

5.3 Frequency-constrained Substring Complexity

For a string X, a dictionary $\mathcal{D}$ of d strings (documents) and a partition of $[d]$ in τ intervals $\mathcal{I} = I_1, \ldots, I_\tau$, the function $f_{X,\mathcal{D},\mathcal{I}}(i, j)$ maps i, j to the number of distinct substrings of length i of X occurring in at least α_j and at most β_j documents in $\mathcal{D}$, where $I_j = [\alpha_j, \beta_j]$. Function f is known as the *frequency-constrained substring complexity* of X [25].

▶ **Example 12.** Let $\mathcal{D} = \{\texttt{a}, \texttt{ananan}, \texttt{baba}, \texttt{ban}, \texttt{banna}, \texttt{nana}\}$. For $X = \texttt{banana}$ and $I_1 = [1, 2], I_2 = [3, 4], I_3 = [5, 6]$, we have $f_{X,\mathcal{D},\mathcal{I}}(2, 2) = 3$: `ba` occurs in $3 \in I_2$ documents; `an` occurs in $4 \in I_2$ documents; and `na` occurs in $3 \in I_2$ documents.

The function $f_{X,\mathcal{D},\mathcal{I}}$ is very informative about X; it provides fine-grained information about the contents (the substrings) of X. It can thus facilitate the tuning of string-processing algorithms by setting bounds on the length or on frequency of substrings; see [25].

Let S be a 2D array such that $S[i, j] = f_{X,\mathcal{D},\mathcal{I}}(i, j)$. Pissis et al. [25] showed that after an $\mathcal{O}(n)$-time preprocessing of a dictionary $\mathcal{D}$ of d strings of total length n over an integer alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$, for any X and any partition $\mathcal{I}$ of $[d]$ in τ intervals given on-line, S can be computed in near-optimal $\mathcal{O}(|X|\tau \log\log d)$ time.

The solution in [25] can be summarized as follows. In the preprocessing step, we construct the generalized suffix tree T of $\mathcal{D}$. In querying, the first step is to construct the suffix tree of X and compute the document frequency of its nodes in $\mathcal{O}(|X|)$ time. In the second step, we enhance the suffix tree of X with $\mathcal{O}(|X|\tau)$ nodes with document frequencies by answering SWA queries on T in $\mathcal{O}(\log\log d)$ time per query [4]. The whole step thus takes $\mathcal{O}(|X|\tau \log\log d)$ time. In the third step, we infer a collection of length intervals, one per node of the enhanced suffix tree and sort them in $\mathcal{O}(|X|\tau)$ time using radix sort. In the last step, we sweep through the intervals from left to right to compute array S in $\mathcal{O}(|X|\tau)$ total time. This concludes the summary of the solution in [25]. We amend the solution as follows.

We plug in Theorem 1 for preprocessing T and for the second step (SWA queries). For SWA queries, as $\mathsf{weight}(u)$, we use the number of dictionary strings having at least one leaf node in the subtree rooted at node u in T. Such an assignment satisfies the requested properties of $\mathsf{weight}(\cdot)$ and can be done in linear time [19]. We obtain the following result.

▶ **Theorem 13.** *For any dictionary $\mathcal{D}$ of d strings of total length n over alphabet $\Sigma = [1, n^{\mathcal{O}(1)}]$, there exists an $\mathcal{O}(n)$-space data structure that answers $S = f_{X,\mathcal{D},\mathcal{I}}$ queries in $\mathcal{O}(|X|\tau)$ time. The preprocessing algorithm runs in $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ space.*

Since S is of size $|X| \cdot \tau$ (it consists of $|X| \cdot \tau$ integers), the complexity bounds are optimal with respect to the preprocessing and query times.

References

1 Paniz Abedin, Arnab Ganguly, Solon P. Pissis, and Sharma V. Thankachan. Efficient data structures for range shortest unique substring queries. *Algorithms*, 13(11):276, 2020. doi:10.3390/A13110276.

2 Stephen Alstrup, Thore Husfeldt, and Theis Rauhe. Marked ancestor problems. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98, November 8-11, 1998, Palo Alto, California, USA*, pages 534–544. IEEE Computer Society, 1998. doi:10.1109/SFCS.1998.743504.

3 Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/S00453-020-00744-0.

4 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Trans. Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

5 Golnaz Badkobeh, Panagiotis Charalampopoulos, Dmitry Kosolobov, and Solon P. Pissis. Internal shortest absent word queries in constant time and linear space. *Theor. Comput. Sci.*, 922:271–282, 2022. doi:10.1016/J.TCS.2022.04.029.

6 Golnaz Badkobeh, Panagiotis Charalampopoulos, and Solon P. Pissis. Internal shortest absent word queries. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.6.

7 Tim Baumann and Torben Hagerup. Rank-select indices without tears. In Zachary Friggstad, Jörg-Rüdiger Sack, and Mohammad R. Salavatipour, editors, *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 85–98. Springer, 2019. doi:10.1007/978-3-030-24766-9_7.

8 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wrocław, Poland*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CPM.2021.8.

9 Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. doi:10.1007/10719839_9.

10 William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994. doi:10.1007/BF01185431.

11 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In Inge Li Gørtz and Oren Weimann, editors, *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020, June 17-19, 2020, Copenhagen, Denmark*, volume 161 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.CPM.2020.8.

12 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal dictionary matching. *Algorithmica*, 83(7):2142–2169, 2021. doi:10.1007/S00453-021-00821-Y.

13 Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, and Wiktor Zuba. Internal quasiperiod queries. In Christina Boucher and Sharma V. Thankachan, editors, *String Processing and Information Retrieval – 27th International Symposium, SPIRE 2020, Orlando, FL, USA, October 13-15, 2020, Proceedings*, volume 12303 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2020. doi:10.1007/978-3-030-59212-7_5.

14 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

15 Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Combinatorial Pattern Matching, 7th Annual Symposium, CPM 96, Laguna Beach, California, USA, June 10-12, 1996, Proceedings*, volume 1075 of *Lecture Notes in Computer Science*, pages 130–140. Springer, 1996. doi:10.1007/3-540-61258-0_11.

16 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993. doi:10.1016/0022-0000(93)90040-4.

17 Pawel Gawrychowski, Moshe Lewenstein, and Patrick K. Nicholson. Weighted ancestors in suffix trees. In Andreas S. Schulz and Dorothea Wagner, editors, *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 455–466. Springer, 2014. doi:10.1007/978-3-662-44777-2_38.

18 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997. doi:10.1017/cbo9780511574931.

19 Lucas Chi Kwong Hui. Color set size problem with application to string matching. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 - May 1, 1992, Proceedings*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer, 1992. doi:10.1007/3-540-56024-6_19.

20 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

21 Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, and Tatiana Starikovskaya. Cross-document pattern matching. *J. Discrete Algorithms*, 24:40–47, 2014. doi:10.1016/J.JDA.2013.05.002.

22 Tsvi Kopelowitz and Moshe Lewenstein. Dynamic weighted ancestors. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 565–574. SIAM, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283444.

23 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5):105:1–105:35, 2021. doi:10.1145/3409371.

24 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In Jon M. Kleinberg, editor, *Proceedings of the 38th Annual ACM Symposium on Theory of Computing, Seattle, WA, USA, May 21-23, 2006*, pages 232–240. ACM, 2006. doi:10.1145/1132516.1132551.

25 Solon P. Pissis, Michael Shekelyan, Chang Liu, and Grigorios Loukides. Frequency-constrained substring complexity. In Franco Maria Nardini, Nadia Pisanti, and Rossano Venturini, editors, *String Processing and Information Retrieval - 30th International Symposium, SPIRE 2023, Pisa, Italy, September 26-28, 2023, Proceedings*, volume 14240 of *Lecture Notes in Computer Science*, pages 345–352. Springer, 2023. doi:10.1007/978-3-031-43980-3_28.

26 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

27 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

28 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973. doi:10.1109/SWAT.1973.13.

29 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.