Ranked Document Selection ⋆

J. Ian Munro[1], Gonzalo Navarro[2], Rahul Shah[3], and Sharma V. Thankachan[1]

[1] Cheriton School of CS, Univ. Waterloo, Canada. {imunro,thanks}@uwaterloo.ca
[2] Dept. of CS, Univ. Chile, Chile. gnavarro@dcc.uchile.cl
[3] School of EECS, Louisiana State Univ., USA. rahul@csc.lsu.edu

Abstract. Let $\mathcal{D}$ be a collection of string documents of n characters in total. The *top-k document retrieval problem* is to preprocess $\mathcal{D}$ into a data structure that, given a query (P, k), can return the k documents of $\mathcal{D}$ most relevant to pattern P. The relevance of a document d for a pattern P is given by a predefined ranking function $w(P, d)$. Linear space and optimal query time solutions already exist for this problem.
In this paper we consider a novel problem, *document selection* queries, which aim to report the kth document most relevant to P (instead of reporting all top-k documents). We present a data structure using $O(n \log^{\epsilon} n)$ space, for any constant $\epsilon > 0$, answering selection queries in time $O(\log k / \log \log n)$, and a linear-space data structure answering queries in time $O(\log k)$, given the locus node of P in a (generalized) suffix tree of $\mathcal{D}$. We also prove that it is unlikely that a succinct-space solution for this problem exists with poly-logarithmic query time.

1 Introduction and Related Work

Document retrieval is a special branch of pattern matching related to information retrieval and web searching. In this problem, the data consists of a collection of text *documents*, and the queries refer to documents rather than text positions [12]. In this paper we focus on arguably the most important of those problems, called *top-k document retrieval*: Given $\mathcal{D} = \{d_1, d_2, d_3, ..., d_D\}$, of total length $n = \sum_{i=1}^{D} |d_i|$, preprocess it into a data structure that, given a pattern P and a threshold k, retrieves the k documents from $\mathcal{D}$ that are more most *relevant* to P, in decreasing order of relevance. The relevance of a document d with respect to P is captured using any function $w(P, d)$ of the starting positions of the occurrences of P in d. A popular example of relevance is the *term frequency* metric, that is, the number of occurrences of P in d. This a well studied problem, and the best known linear space data structure can answer queries in optimal time $O(k)$ [17], once the locus node of P in a generalized suffix tree of $\mathcal{D}$ is found.

In this paper we study a new related problem called *document selection*, where we must return the kth document of $\mathcal{D}$ most relevant to P, that is, the kth element returned by a top-k query (breaking ties arbitrarily).

⋆ Funded in part by NSERC of Canada and the Canada Research Chairs program, Fondecyt Grant 1-140796, Chile, and NSF Grants CCF–1017623, CCF–1218904

We present three results, depending on the amount of space used: (1) We give a data structure that uses $O(n \log^{\epsilon} n)$ space, for any constant $\epsilon > 0$, and answers queries in time $O(\log k / \log \log n)$. (2) We give a linear-space data structure that answers queries in $O(\log k)$ time. (3) We prove that it is highly unlikely that the problem can be solved in less than linear space within poly-logarithmic time, via a reduction from the *position restricted substring searching* problem [9, 5].

Document selection is useful for various advanced queries. When a user browses ranked results of a query and asks for the next set of results, we need to report the top-k_2 documents that are not top-k_1. Instead of computing a top-k_2 query in time $O(k_2)$, which is nonoptimal if $k_2 - k_1 = o(k_2)$, our results allow solving this query in $O((k_2 - k_1) \log k_2)$ time and linear space. Another possible query is to count the number K of documents d with $w(P, d) \geq \tau$, given P and τ. This can be answered via doubling search using document selection queries, in time $O(\log^2 K)$, assuming $w(P, d)$ can be computed in constant time given the locus of P. Similarly, we can count or list the documents d with $w(P, d) \in [\tau_1, \tau_2]$. Such queries are important in bioinformatics, for example for motif mining or for avoiding sequences where P is "over-expressed", and for data mining in general, for example to estimate the distribution of relevance scores of certain patterns.

Related Work. The notion of relevance-based string retrieval was introduced by Muthukrishnan [11], who proposed and solved various problem but not top-k document retrieval. The first data structure for this problem, under the term frequency measure and using $O(n \log n)$ words of space, was given by Hon et al. [4]. Later, Hon et al. [6] introduced a linear space structure ($O(n)$ words), that works for general weight functions as described earlier, with query time $O(p + k \log k)$. This was improved to $O(p + k)$ [13], and finally to the optimal $O(k)$ [17], all using linear space. Those times are in addition to the time for finding the locus node of P, $\texttt{locus}(P)$, in the generalized suffix tree of $\mathcal{D}$, GST.

The problem has also been studied in scenarios where less than linear space (i.e., $o(n \log n)$ *bits*) can be used. For example, it is possible to solve the problem efficiently using $n \log \sigma + o(n \log \sigma)$ bits [14, 18], where σ is the alphabet size of the text (thus $n \log \sigma$ bits are used to represent the text itself). The results are mostly tailored to the term frequency measure of relevance, and achieve times of the form $O(k \operatorname{polylog} n)$. See [12, 3, 7] for more details.

2 The top-*k* Framework

This section briefly describes the linear-space framework of Hon et al. [6] for *top-k* queries. The generalized suffix tree (GST) of a document collection $\mathcal{D} = \{d_1, d_2, d_3, \ldots, d_D\}$ is the combined compact trie of all the non-empty suffixes of all the documents [19]. The total number of leaves in GST is same as the total length n of all the documents. For each node j in GST, $prefix(j)$ is the string obtained by concatenating the edge labels on the path from the root to node j. The highest node v satisfying that P is a prefix of $prefix(v)$ is called the *locus* of P and denoted $\texttt{locus}(P) = v$.

Let ℓ_i represent the ith leftmost leaf node in GST. We say that a node is *marked* with a document d if it is either a leaf node whose corresponding suffix belongs to d, or it is the lowest common ancestor (LCA) of two such leaves. This implies that the number of nodes marked with document d is exactly equal to the number of nodes in the suffix tree of d (at most $2|d|$). A node can be marked with multiple documents. For each node j and each of its marking documents d, define a *link* to be a quadruple $(origin = j, target, doc = d, weight = w(prefix(j), d))$, where *target* is the lowest proper ancestor of node j marked with d (a dummy parent of the root node is added, marked with all the documents). Since the number of links with document $doc = d$ is at most $2|d|$, the total number of links is $\leq \sum_{i=1}^{D} 2|d_i| \leq 2n$. The following is a crucial observation by Hon et al. [6].

Lemma 1 *For each document d that contains a pattern P, there is a unique link with* origin *in the subtree of* $\mathtt{locus}(P)$*, a proper ancestor of* $\mathtt{locus}(P)$ *as its* target*, and* weight $w(P, d)$*.*

We say that a link is *stabbed* by a node j if its origin is in the subtree of j (j itself included) and its target is a proper ancestor of j. Therefore, the problem of finding the kth most relevant document for P can be reduced to finding the kth highest weighted link stabbed by $\mathtt{locus}(P)$.

3 Super-Linear Space Structure

In this section we start by introducing a basic data structure that uses $O(n \log n)$ words and answers queries in $O(\log n)$ time. Then we enhance it to a structure that uses $O(n \log^{1+\epsilon} n)$ words, for any constant $\epsilon > 0$, and $O(\log n / \log \log n)$ time. The basic structure will be used in Section 4 to achieve linear space within the same time, whereas the enhanced one will be reduced to $O(n \log^{\epsilon} n)$ words. In Section 5 we show how how the linear-space structure can be improved to answer queries in time $O(\log k)$ and the enhanced structure in time $O(\log k / \log \log n)$, thus reaching our final results.

3.1 The Basic Structure

We prove the following result.

Lemma 2 *Given the* GST *of a text collection of total length n, we can build an $O(n \log n)$-word structure that, given* $\mathtt{locus}(P)$ *and k, answers the document selection query in time $O(\log n)$.*

Let N represent the set of nodes in GST and S represent the set of links $(origin, target, doc, weight)$ in GST, as described in Section 2. Next we construct a balanced binary tree $\mathcal{T}$ of $|S|$ leaves, so that the ith highest weighted link (ties broken arbitrarily) is associated with the ith leftmost leaf of $\mathcal{T}$. Notice that $n \leq |S| \leq 2n$. We use $S(x)$ to denote the set of links associated with the leaves in the subtree of node $x \in \mathcal{T}$. Further, let $N(x)$ denote the set of nodes in GST

that are (i) either the origin or the target of a link in $S(x)$, or (ii) the LCA of two such nodes. Clearly $|N(x)| = \Theta(|S(x)|) = \Theta(n/2^{depth(x)})$, where $depth(x)$ is the number of ancestors of x (depth of root is 0).

With every node $x \in \mathcal{T}$, we associate a tree structure $\mathsf{GST}(x)$. $\mathsf{GST}(x)$ is the subtree of GST obtained by retaining only the nodes in $N(x)$, so that node v is the parent of node w in $\mathsf{GST}(x)$ iff v is the lowest proper ancestor of w in GST that also belongs to $N(x)$. The number of nodes and edges in $\mathsf{GST}(x)$ is $\Theta(n/2^{depth(x)})$.

Notice that the same node $w \in \mathsf{GST}$ may appear in several $\mathsf{GST}(\cdot)$'s. With each node $w \in \mathsf{GST}(x)$ we associate the following information:

- $stab.count_x(w)$: The number of links in $S(x)$ that are stabbed by w.
- $left.ptr_x(w)$: Let x_L be the left child of x (in $\mathcal{T}$). Let w_L be the highest node in the subtree of w (in $\mathsf{GST}(x)$) that appears also in $\mathsf{GST}(x_L)$ (w_L can be w itself). Then $left.ptr_x(w)$ is a pointer from $w \in \mathsf{GST}(x)$ to $w_L \in \mathsf{GST}(x_L)$. If there exists no such node w_L, then $left.ptr_x(w)$ is null.
- $right.ptr_x(w)$: Analogous to $left.ptr_x(w)$, now considering x_R, the right child of $x \in \mathcal{T}$, and w_R being the highest node in the subtree of $w \in \mathsf{GST}(x)$ that appears also in $\mathsf{GST}(x_R)$.

Note that the space needed for maintaining $\mathsf{GST}(x)$ and the associated information is $O(n/2^{depth(x)})$ words. Added over all the nodes $x \in \mathcal{T}$, the total space occupancy of all $\mathsf{GST}(\cdot)$'s is $O(n \log n)$ words. Finally, the following result is crucial for our data structure (the case of w_R and x_R is analogous).

Lemma 3 *Both w and w_L stab the same subset of links of $S(x_L)$.*

Proof. Otherwise, the target of a link in $S(x_L)$ stabbing w_L but not w would be higher than w_L, below w, and belong to $\mathsf{GST}(x_L)$, contradicting the definition of w_L. The same happens with the source of a link stabbing w but not w_L. □

3.2 Query Algorithm for Document Selection

Assume $\texttt{locus}(P)$ is given. Notice that the tree $\mathsf{GST}(root)$ associated with the *root* of $\mathcal{T}$ is the same GST of the collection. Therefore, $stab.count_{root}(\texttt{locus}(P))$ gives the number of documents containing P. If the count is less than k, there is no kth document to select. Otherwise, let L^* be the kth highest weighted link stabbed by $\texttt{locus}(P)$. Our query algorithm traverses $\mathcal{T}$ top-down, starting from *root* and ending at the leaf node associated with link L^*. Then it reports the document d^* corresponding to L^*.

In our query algorithm, we use x to denote a node in $\mathcal{T}$, w to denote a node in $\mathsf{GST}(x)$ and K to denote an integer $\leq k$. First we initialize x to the root of $\mathcal{T}$, w to $\texttt{locus}(P)$ and K to k. This establishes the invariant that we have to return the Kth highest weighted link in $S(x)$ stabbed by w. Let x_L and x_R be the left and right children of x. Then we obtain the nodes $w_L \in \mathsf{GST}(x_L)$ and $w_R \in \mathsf{GST}(x_R)$ pointed by $left.ptr_x(w)$ and $right.ptr_x(w)$, respectively. The following values are then computed in constant time.

– $c = stab.count_x(w)$, the number of links in $S(x)$ stabbed by w.
– $c_L = stab.count_{x_L}(w_L)$, the number of links in $S(x_L)$ stabbed by w (or w_L).
– $c_R = stab.count_{x_R}(w_R)$, the number of links in $S(x_R)$ stabbed by w (or w_R).

Notice that $c = c_L + c_R$. If $c_L \geq K$ then, by Lemma 3, the Kth link below $S(x)$ (or $S(x_L)$) stabbed by $w \in \mathsf{GST}(x)$ is the same as the Kth link below $S(x_L)$ stabbed by $w_L \in \mathsf{GST}(x_L)$. Therefore, we maintain the invariant if we continue the traversal in the subtree of $x \leftarrow x_L$ with $\mathsf{GST}(x_L)$ node $w \leftarrow w_L$. On the other hand, if $c_L < K$, then by Lemma 3 the Kth link stabbed by w below $S(x)$ is same as the $(K - c_L)$th link below $S(x_R)$ stabbed by $w_R \in \mathsf{GST}(x_R)$. In this case, we maintain the invariant if we continue the traversal in the subtree of $x \leftarrow x_R$ with $\mathsf{GST}(x_R)$ node $w \leftarrow w_R$ and with $K \leftarrow K - c_L$. We terminate the algorithm when x is a leaf, thus $K = 1$ and x represents L^*. As the height of $\mathcal{T}$ is $O(\log n)$ and the time spent at each node is constant, the total query time is $O(\log n)$ and Lemma 2 is proved.

3.3 An Enhanced Structure

We now prove the following result, which will hold in the RAM model of computation, with a computer word of $w = \Omega(\log n)$ bits.

Lemma 4 *Given the* GST *of a text collection of total length* n *and any constant* $0 < \epsilon \leq 1$*, we can build an* $O(n \log^{1+\epsilon} n)$*-word structure that, given* $\mathtt{locus}(P)$ *and* k*, answers the document selection query in time* $O(\log n / \log\log n)$*.*

In order to speed up the structure of Lemma 2, we will choose a step $s = \epsilon \log\log n$ and build the $\mathsf{GST}(x)$ structures only for nodes $x \in \mathcal{T}$ whose depth is a multiple of s. Each node $w \in \mathsf{GST}(x)$ for the selected nodes x will store sufficient information for the query algorithm to jump directly to the corresponding node x' at depth $depth(x') = depth(x) + s$, instead of just to x_L or x_R.

Given $x, x' \in \mathcal{T}$ as above (x' in the subtree of x) and $w \in \mathsf{GST}(x)$, we define $w_{x'}$ as the highest node in the subtree of w that appears also in $\mathsf{GST}(x')$. Let us call $x_1, x_2, \ldots, x_{2^s}$ the nodes at depth $depth(x) + s$ that descend from x (or the leaves below x, if they have depth less than $depth(x) + s$), ordered left to right in $\mathcal{T}$ (i.e., from highest to lowest weights in $S(x_i)$).

Associated to each node $w \in \mathsf{GST}(x)$, we store 2^s pointers $ptr_x(w)[i] = w_{x_i}$. We also store the 2^s cumulative values $acc_x(w)[i] = \sum_{j=1}^{i} stab.count_{x_j}(w_{x_j})$; note that $acc_x(w)[2^s] = stab.count_x(w)$. We will store those $acc_x(w)$ values in a fusion tree [1], which takes $O(2^s) = O(\log^\epsilon n)$ words of space and solves predecessor queries in $acc_x(w)$ in constant time. The space is the same used by array $ptr_x(w)$, which added over all the $\mathsf{GST}(\cdot)$'s is $O(n \log^{1+\epsilon} n)$ words (even if only one level out of s in $\mathcal{T}$ stores $\mathsf{GST}(\cdot)$ structures).

Queries now proceed as in Section 3.2, but now we use the fusion tree to determine, given $w \in \mathsf{GST}(x)$, which is the node $x_i \in \mathcal{T}$ that contains the Kth link below $S(x)$ stabbed by w. Therefore we can move directly from x to x_i and from $w \in \mathsf{GST}(x)$ to $w_i \in \mathsf{GST}(x_i)$, where $w_i = ptr_x(w)[i]$. We also update $K \leftarrow K - acc_x(w)[i-1]$ (assume $acc_x(w)[0] = 0$). Thus we complete the query in $O((\log n)/s) = O(\log n / (\epsilon \log\log n))$ constant-time steps and Lemma 4 is proved.

4 Linear Space Structure

In this section we build on the basic structure of Lemma 2 in order to achieve linear space and logarithmic query time. At the end, we reduce the space of the enhanced structure to $O(n \log^{\epsilon} n)$. The results hold under the RAM model.

Lemma 5 *Given the* GST *of a text collection of total length* n*, we can build an* $O(n)$*-word structure that, given* $\mathtt{locus}(P)$ *and* k*, answers the document selection query in time* $O(\log n)$*.*

To achieve linear space, we replace some of our data structures by succinct ones. We will measure the space in bits, aiming at using $O(n \log n)$ bits overall. The binary tree $\mathcal{T}$ can be maintained in $O(n \log n)$ bits, where each internal node x stores an $O(\log n)$-bit pointer to the corresponding tree $\mathsf{GST}(x)$ and each leaf stores the document identifier corresponding to the associated link. The global GST can also be maintained in $O(n \log n)$ bits. Therefore, the space-consuming component are the $\mathsf{GST}(\cdot)$'s and their associated information.

Using well-known succinct data structures [16], the $\mathsf{GST}(x)$ tree topologies can be represented in $O(1)$ bits per node (i.e., $O(n \log n)$ bits overall) with constant-time support of all the basic navigational operations required in our algorithm. We refer to any node $w \in \mathsf{GST}(x)$ by its pre-order rank, that is, node j means the node with pre-order rank j. The pre-order rank of the root node of any $\mathsf{GST}(x)$ is 1. Next we show how to encode the remaining information associated with each node in $\mathsf{GST}(x)$ using $O(1)$ bits per node.

4.1 Encoding $stab.count_x(j)$

We note that $stab.count_x(j)$ is exactly equal to the number of links of $S(x)$ associated with $\mathsf{GST}(x)$ that originate in the subtree of j minus the number of links in $S(x)$ that target any node in the subtree of j (j belongs to its subtree). We encode this information in two bit vectors: $B_x = 10^{\alpha_1}10^{\alpha_2}10^{\alpha_3}\ldots$ and $B'_x = 10^{\beta_1}10^{\beta_2}10^{\beta_3}\ldots$, where α_j (resp., β_j) is the number of links of $S(x)$ originating from (resp., targeting at) node j in $\mathsf{GST}(x)$. We augment B_x and B'_x with structures supporting constant-time rank/select queries [10]. Notice that $\sum \alpha_j = \sum \beta_j = O(|S(x)|) = O(|\mathsf{GST}(x)|)$. Therefore, both B_x and B'_x can be represented in $O(1)$ bits per node.

Now we can compute $stab.count_x(j)$ for any j in $O(1)$ time as follows: find the rightmost leaf node j' in the subtree of j in $O(1)$ time using the succinct tree representation of $\mathsf{GST}(x)$ [16]. Then the number n_o of links originating from the subtree of j is equal to the number of 0-bits between the jth and $(j'+1)$th 1-bit in B_x (because j and j' are preorder numbers). Similarly, the number n_t of links targeted at any node in the subtree of j is equal to the number of 0-bits between the jth and $(j'+1)$th 1-bits in B'_x. Using rank/select operations on B_x and B'_x, n_o and n_t are computed in $O(1)$ time and $stab.count_x(j)$ is given by $n_o - n_t$.

4.2 Encoding *left.ptr*$_x(j)$ and *right.ptr*$_x(j)$

We show how to encode *left.ptr*$_x(\cdot)$ for all nodes in $\mathsf{GST}(x)$; *right.ptr*$_x(j)$ is symmetric. The idea is to maintain a bit vector LP such that $LP[j] = 1$ iff there exists a node $j_L \in \mathsf{GST}(x_L)$ such that both $j \in \mathsf{GST}(x)$ and $j_L \in \mathsf{GST}(x_L)$ represent the same node in GST. We add constant-time rank/select data structures [10] on LP. Since the length of LP is equal to the number of nodes in $\mathsf{GST}(x)$, its space occupancy is $O(1)$ bits per node.

Now, for any given node $j \in \mathsf{GST}(x)$, the node $j_L \in \mathsf{GST}(x_L)$ to which *left.ptr*$_x(j)$ points is the (unique) highest descendant of j that is marked in LP, thus it can be identified by (1) finding the position j^* of the leftmost 1-bit in $LP[j\ldots]$; (2) checking if node j^* is in the subtree of node j in $\mathsf{GST}(x)$; (3) if so, then $j_L \in \mathsf{GST}(x_L)$ is equal to the number of 1's in $LP[1...j^*]$, otherwise, j_L is null. All these operations require constant time, either using the succinct tree operations or the rank/select data structures. This works because all the nodes in $\mathsf{GST}(x_L)$ appear in $\mathsf{GST}(x)$, in the same order (pre-order).

In summary, the space requirement of our encoding scheme is $O(1)$ bits per node in any $\mathsf{GST}(x)$, thus adding to $O(n \log n)$ bits. The query algorithm, as well as its time complexity, remain the same. This completes the proof of Lemma 5.

4.3 Reducing Space of the Enhanced Structure

The space of the enhanced structure of Section 3.3 can be similarly reduced to $O(n \log^\epsilon n)$ words, obtaining the following result.

Lemma 6 *Given the* GST *of a text collection of total length* n *and a constant* $\epsilon > 0$*, we can build an* $O(n \log^\epsilon n)$*-word structure that, given* $\mathtt{locus}(P)$ *and* k*, answers the document selection query in time* $O(\log n / \log\log n)$*.*

For this sake, recalling the definition of $x_1, \ldots, x_{2^s}$ of Section 3.3, we will maintain bit vectors LP_i for $i = 1$ to 2^s, so that $LP_i[j] = 1$ iff there exists a node $j_i \in \mathsf{GST}(x_i)$ such that both $j \in \mathsf{GST}(x)$ and $j_i \in \mathsf{GST}(x_i)$ represent the same node in GST. Then each array entry $ptr_x(j)[i]$ is computed using LP_i as in Section 4.2. The total space used by all the LP_i bit vectors is $O(2^s) = O(\log^\epsilon n)$ bits per node, adding up to $O(n \log^{1+\epsilon} n)$ bits in total.

To compute $acc_x(j)[i]$, we store bitmaps $B_{x,1}, \ldots, B_{x,2^s}$ and $B'_{x,1}, \ldots, B'_{x,2^s}$, analogous to B and B' of Section 4.1. In this case, $B_{x,i} = 10^{\alpha_1^i} 10^{\alpha_2^i} 10^{\alpha_3^i} \ldots$, so that $\alpha_j^i = \sum_{r=1}^{i} s(r)$, where $s(r)$ is the number of links of $S(x_r)$ originating from node $ptr_x(j)[i] \in \mathsf{GST}(x_r)$, and $B'_{x,i} = 10^{\beta_1^i} 10^{\beta_2^i} 10^{\beta_3^i} \ldots$, so that $\beta_j^i = \sum_{r=1}^{i} t(r)$, where $t(r)$ is the number of links of $S(x_r)$ targeting at node $ptr_x(j)[i] \in \mathsf{GST}(x_r)$. Then, it holds $acc_x(j)[i] = \alpha_j^i - \beta_j^i$, which is computed in constant time using rank/select operations. Since it holds $\alpha_j^i \leq \alpha_j$ and $\beta_j^i \leq \beta_j$ for all i values, the total space of these $2^s = \log^\epsilon n$ bitmaps adds up to $O(n \log^{1+\epsilon} n)$ bits.

To carry out predecessor searches on the virtual vector $acc_x(j)$, we use succinct SB-trees [2, Lemma 3.3]. Given constant-time access to any $acc_x(j)[i]$, this structure provides predecessor searches in $O(1 + \log(2^s) / \log\log n) = O(1)$ time and use $O(2^s \log\log n) = O(\log^\epsilon n)$ bits per node (by adjusting ϵ). Thus the total space is $O(n \log^{1+\epsilon} n)$ bits as well. This concludes the proof of Lemma 6.

5 Achieving $O(\log k)$ Query Time and Better

In this section we first build on the linear-space data structure of Lemma 5 in order to improve its query time to $O(\log k)$. At the end, we show that the result extends to our superlinear-space data structure of Lemma 6, improving its query time to $O(\log k / \log\log n)$. Thus we start by proving the following theorem.

Theorem 1 *A collection $\mathcal{D}$ of documents can be preprocessed into a linear-space data structure that can answer any* document selection *query (P, k) in time $O(\log k)$, given the locus of pattern P in the generalized suffix tree of $\mathcal{D}$.*

Notice that the query time $O(\log n)$ in Lemma 5 can be written as $O(\log k)$ for $k > \sqrt{n}$. Therefore, we turn our attention to the case where $k \leq \sqrt{n}$. First, we derive a space-efficient structure $DS(\delta)$, which can answer document selection queries faster, but only for values of k below a predefined parameter $\delta \leq \sqrt{n}$. More precisely, structure $DS(\delta)$ will satisfy the following properties:

Lemma 7 *The structure $DS(\delta)$ uses $O(n(\log \delta + \log\log n))$ bits of space and can answer document selection queries in time $O(\log \delta + \log\log n)$, for $k \leq \delta \leq \sqrt{n}$.*

To obtain the result in Theorem 1, we maintain structures $DS(\delta_i)$ with $\delta_i = \lceil n^{1/2^i} \rceil$ for $i = 1, 2, 3, \ldots, r$, where $\delta_{r+1} \leq \sqrt{\log n} < \delta_r$ (therefore $r < \log\log n$). The total space needed is $O(n \sum_{i=1}^{r} (\log \delta_i + \log\log n)) = O(n \log n)$ bits ($O(n)$ words). When k comes as a query, if $k > \delta_{r+1}$, we first find h, where $\delta_{h+1} < k \leq \delta_h$ and obtain the answer using $DS(\delta_h)$. The resulting time is $O(\log \delta_h + \log\log n) = O(\log k)$. The case where $k < \delta_{r+1}$ is handled separately using other structures in $O(1)$ time (Section 5.2). We now describe the details of $DS(\delta)$.

5.1 Structure $DS(\delta)$

The first step is to identify certain nodes in GST as *marked* nodes and *prime* nodes, based on a parameter $g = \lceil \delta \log n \rceil$ called the *grouping factor*. Every gth leftmost leaf is marked, and the LCA of every two consecutive marked leaves is also marked. Therefore, the number of marked nodes is $\Theta(n/g)$. Nodes with their parent marked are prime. A prime node with at least one marked node in its subtree is a type-1 prime node, otherwise it is a type-2 prime node. Notice that the highest marked node in the subtree of any node is unique, if it exists. Therefore, except the root node, every marked node j^* can be associated with a unique type-1 prime node j', which is the first prime node on the path from j^* to the root. Notice that a node can be both prime and marked.

Let j' be a prime node and j^* be the highest marked node in its subtree (j^* exists only if j' is of type-1, and it can be that $j' = j^*$). We use $G(j' \backslash j^*)$ to represent the subtree of GST rooted at j' after removing the subtree of j^* (j^* is not removed). With a slight abuse of notation, we use $G(j' \backslash j^*)$ to represent the set of nodes within $G(j' \backslash j^*)$ as well. A crucial result [17] is that, for any prime node j', the number of nodes in $G(j' \backslash j^*)$ is $O(g)$.

We define *prime.parent*(j) of any node j in GST as the first prime node j' on the path from j to the root. Note that $j \in G(j' \backslash j^*)$, otherwise j would be a (strict) descendant of j^* and its corresponding j' would be below j^*.

It is not hard to determine $j' = prime.parent(j)$ in constant time and $O(n)$ bits, by sampling the prime nodes in a succinct tree representation and looking for the lowest sampled ancestor of j [15, Lemma 4.4].

The structure $DS(\delta)$ is a collection of substructures $STR(j')$ associated with every prime node j' in GST. If the input node $\texttt{locus}(P) \in G(j' \backslash j^*)$ and $k \leq \delta$, we obtain the answer using $STR(j')$ in $O(\log g) = O(\log \delta + \log \log n)$ time. Based on the type of j', we have two cases; we describe the simpler one first.

***STR*(*j*′) associated with a type-2 prime node *j*′:** The structure can be constructed as follows: take $G(j')$, the subtree rooted at node j', and replace the pre-order rank of each node j by $(j-j'+1)$. Also associate a dummy parent node to the root. Then, among the links defined over GST (Section 2), choose those that originate from the subtree of j' and: (1) Assign a new value to its origin and target, which is its original value minus j' plus 1. The target of some links can be negative; replace those by 0. (2) Replace the weight by a rank-space reduced value in $[1, O|G(j')|]$. Notice that the number of links chosen is $O(|G(j')|)$. (3) Let d be its document identifier. Instead of writing d explicitly in $\lceil \log D \rceil$ bits, use a pointer to one leaf node in $G(j')$, using $\lceil \log |G(j')| \rceil$ bits, where the suffix corresponding to that leaf belongs to document d.

In summary, we have a tree of $(|G(j')| + 1)$ nodes and $O(|G(j')|)$ links associated with it. The information $(origin, target, document, weight)$ associated with each link is encoded in $O(\log |G(j')|)$ bits. Then $STR(j')$ is the structure described in Lemma 5 over these nodes and links. The space required is $O(|G(j')| \log |G(j')|) = O(|G(j')| \log g)$ bits. We maintain structures $STR(j')$ for all type-2 prime nodes j' in total $O(n \log g)$ bits, since a node can be in the subtree of at most one type-2 prime node.

***STR*(*j*′) associated with a type-1 prime node *j*′:** We first identify the *candidate set* $\mathcal{C}(j')$ of $O(g)$ links, such that for any $k \leq \delta$, the kth link stabbed by any node $j \in G(j' \backslash j^*)$ belongs to $\mathcal{C}(j')$. Clearly we can ignore the links that do not originate from the subtree of j'. The links that do can be categorized into the following types [17]: *near-links* are stabbed by j^*, but not by j'; *far-links* are stabbed by both j^* and j'; *small-links* are targeted at a node in the subtree of j^*; and *fringe-links* are the others.

We include all near-links and fringe-links into $\mathcal{C}(j')$, which are $O(g)$ in number [17, Lemma 8]. All small-links can be ignored as none of them is stabbed by any node in $G(j' \backslash j^*)$. Notice that if any node in $G(j' \backslash j^*)$ stabs a far-link, it indeed stabs all far-links. Therefore, it is sufficient to insert the top-δ far-links into $\mathcal{C}(j')$. Thus, we have $O(g)$ links in $\mathcal{C}(j')$ overall.

Now we perform a rank-space reduction of pre-order rank of nodes in $G(j' \backslash j^*)$ as well as of the information associated with the links in $\mathcal{C}(j')$, as follows:

- The target of those links targeting at any proper ancestor of j' is changed to a dummy parent node of j'. Similarly, the origin of all those links originating in the subtree of j^* is changed to node j^*.
- The pre-order rank of all those nodes in $G(j' \backslash j^*)$, and the corresponding origin and target values of links in $\mathcal{C}(j')$, are changed to a rank-space reduced value in $[0, |G(j' \backslash j^*)|]$. Notice that the new pre-order rank of j' is 1 and that of its dummy parent node is 0. We remark that this mapping (and remapping) can be stored separately in $O(|G(j' \backslash j^*)| \log |G(j' \backslash j^*)|)$ bits.
- The weights of the links are also replaced by rank-space reduced values.
- Let L be a near- or fringe-link in $\mathcal{C}(j')$ with d its corresponding document. Then there must be at least one leaf ℓ in $G(j' \backslash j^*)$ where the suffix corresponding to ℓ belongs to d. Therefore, instead of representing d, we maintain a pointer to ℓ, which takes only $O(\log g)$ bits. This trick will not work for far-links, as the existence of such a leaf node is not guaranteed. Therefore, we spend $\log D$ bits for each far-link, which is still affordable because there are only $O(\delta) = O(g/\log n)$ far-links.

In summary, we have a tree of $(|G(j' \backslash j^*)| + 1) = O(g)$ nodes with $O(g)$ links associated with it. Then $STR(j')$ is the structure described in Lemma 5 over these nodes and links. The space required is $O(g \log g)$ bits. As the number of type-1 prime nodes is $O(n/g)$, the total space to maintain $STR(j')$ for all type-2 primes nodes j' is $O(n \log g)$ bits.

Query Answering: Given node $j = \texttt{locus}(P)$, we find $j' = \textit{prime.parent}(j)$. Then we map node j to the corresponding node in $STR(j')$ and obtain the answer by querying $STR(j')$, in $O(\log g) = O(\log \delta + \log \log n)$ time. The answer may come in the form of a node in $STR(j')$, which is mapped back to GST in order to obtain the associated document. This completes the proof of Lemma 7.

5.2 Structure for $k \le \delta_{r+1}$

First, identify the marked and prime nodes in GST with $g = \delta_{r+1} \log n$. At every prime node j', we explicitly maintain the candidate set $\mathcal{C}(j')$. This takes $O(n)$-word space. Then for any $k \le \delta_{r+1}$, the kth link stabbed by node j can be encoded as a pointer to the corresponding entry in $\mathcal{C}(\textit{prime.parent}(j'))$ using $\lceil \log |\mathcal{C}(\textit{prime.parent}(j'))| \rceil = O(\log g) = O(\log \log n)$ bits. Therefore, the answers for all $k \in [1, \delta_{r+1}]$ for all nodes in GST can be maintained in additional $O(n \cdot \delta_{r+1} \log \log n) = o(n \log n)$ bits of space. Now the kth link (and its document) stabbed by any query node $\texttt{locus}(P)$ can be obtained from $\mathcal{C}(\textit{prime.parent}(\texttt{locus}(P)))$ in $O(1)$ time.

5.3 Speeding Up the Enhanced Structure

The same construction used above can be used to speed up our superlinear-space structure of Lemma 6, simply by using it instead of the linear-space one of Lemma 5 to implement the structures $STR(j')$. The space of the form $O(n \log^{\epsilon} n)$ words, or $O(n \log^{1+\epsilon} n)$ bits, will become $O(g \log g \log^{\epsilon} n)$ inside the structures $STR(j')$, because we will maintain the sampling step $s = \epsilon \log\log n$ depending on n, not on g, and use the succinct SB-trees with parameter n, not g. As a result, the total space per value of δ will be $O(n \log g \log^{\epsilon} n)$ bits, and added over all the values of δ we will have $O(n \log^{\epsilon} n \sum_{i=1}^{r} (\log \delta_i + \log\log n)) = O(n \log^{1+\epsilon} n)$ bits, or $O(n \log^{\epsilon} n)$ words. The time, on the other hand, will be $O(1 + \log \delta / (\epsilon \log\log n))$ on $DS(\delta)$, which becomes $O(1 + \log k / (\epsilon \log\log n))$ in terms of k. We have proved our final result for the superlinear structure.

Theorem 2 *A collection $\mathcal{D}$ of documents of total length n can be preprocessed into a data structure using $O(n \log^{\epsilon} n)$ words of space, for any constant $\epsilon > 0$, which can answer* document selection *queries (P, k) in time $O(1+\log k / \log\log n)$, given the locus of pattern P in the generalized suffix tree of $\mathcal{D}$.*

6 Hardness of an Efficient Succinct Solution

One could expect to obtain an index using $O(n \log \sigma)$ bits of space, proportional to the $n \log \sigma$ bits needed to store $\mathcal{D}$, as achieved for the top-k document retrieval problem. We show, however, that this is very unlikely unless a significant breakthrough in the current state of the art of computational geometry is obtained.

Theorem 3 *If there exists a data structure using $O(n \log \sigma + D \,\mathrm{polylog}\, n)$ bits with query time $O(|P| \,\mathrm{polylog}\, n)$ for document selection (σ being the alphabet size), then there exists a linear-space data structure that can answer three-dimensional range reporting queries in poly-logarithmic time per reported point.*

Proof. We reduce from the position restricted substring searching (PRSS) problem, which is defined as follows: Index a given a text $T[1, n]$ over an alphabet set $[1, \sigma]$, such that whenever a pattern P (of length p) and a range $[x, y]$ comes as a query, all those $occ_{x,y}$ occurrences of P in $T[x \ldots y]$ can be reported efficiently. Many indexes offering different space and query time trade-offs exist [9, 8].

Hon et al. [5] proved that answering PRSS queries in polylog time and succinct space is at least as hard as performing 3-dimensional orthogonal range reporting in polylog time and linear space. They also showed that if the query pattern is longer than $\alpha = \lceil \log^{2+\epsilon} n \rceil$ for some predefined constant $\epsilon > 0$, an efficient succinct space index can be designed. Therefore, the harder case arises when $p < \alpha$. We now show how to answer PRSS queries with $p < \alpha$ via document selection queries on the following set: $\mathcal{D} = \{d_1, d_2, d_3, ..., d_{\lceil n/\alpha \rceil}\}$, where $d_i = T[1 + (i-1)\alpha ... (i+1)\alpha]$ and $|d_i| = 2\alpha$, except possibly for $d_{\lceil n/\alpha \rceil - 1}$ and $d_{\lceil n/\alpha \rceil}$. The score function $w(P, d_i)$ is i if P appears at least once in d_i and 0 otherwise. Notice that an occurrence of any pattern of length at most α overlaps with at least one and at most two documents in $\mathcal{D}$. Therefore, the previously defined PRSS query on T can be answered via multiple document selection queries on $\mathcal{D}$ as follows: first report all those documents d_i with $w(P, d_i) \in [\lceil x/\alpha \rceil, \lfloor y/\alpha + 2 \rfloor]$. Then, within all those reported documents, look for other occurrences of P via an exhaustive scanning. If the time for document selection queries is polylog in the total length of all documents in $\mathcal{D}$ (which is at most $2n$), then the time for PRSS query is also bounded by $O((p + occ_{x,y})\text{polylog}\, n)$. Therefore, answering document selection queries in polylog time and succinct space is at least as hard as answering PRSS queries in polylog time and succinct space. $\square$

References

1. M. Fredman and D. Willard. Surpassing the information theoretic barrier with fusion trees. *J. Comp. Sys. Sci.*, 47:424–436, 1993.
2. R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *STACS*, pages 517–528, 2009.
3. W.-K. Hon, M. Patil, R. Shah, S. V. Thankachan, and J. S. Vitter. Indexes for document retrieval with relevance. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 351–362, 2013.
4. W.-K. Hon, M. Patil, R. Shah, and S.-B. Wu. Efficient index for retrieving top-k most frequent documents. *J. Discr. Alg.*, 8(4):402–417, 2010.
5. W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. On position restricted substring searching in succinct space. *J. Discr. Alg.*, 17:109–114, 2012.
6. W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722, 2009.
7. W.-K. Hon, S. V. Thankachan, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval. *J. of the ACM*, 2014. To appear.
8. M. Lewenstein. Orthogonal range searching for text indexing. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 267–302. Springer, 2013.
9. V. Mäkinen and G. Navarro. Position-restricted substring searching. In *LATIN*, pages 703–714, 2006.
10. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.
11. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.
12. G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. *ACM Computing Surveys*, 46(4):article 52, 2014.
13. G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *SODA*, pages 1066–1077, 2012.
14. G. Navarro and S. V. Thankachan. Faster top-k document retrieval in optimal space. In *SPIRE*, pages 255–262, 2013.
15. L. Russo, G. Navarro, and A. Oliveira. Fully-compressed suffix trees. *ACM Trans. Alg.*, 7(4):art. 53, 2011.
16. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA*, pages 134–149, 2010.
17. R. Shah, C. Sheng, S. V. Thankachan, and J. S. Vitter. Top-k document retrieval in external memory. In *ESA*, pages 803–814, 2013.
18. D. Tsur. Top-k document retrieval in optimal space. *Inf. Process. Lett.*, 113(12):440–443, 2013.
19. P. Weiner. Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11, 1973.