Efficient Algorithms for Document Retrieval Problems

S. Muthukrishnan*

Abstract

We are given a collection D of text documents $d_1, \ldots, d_k$, with $\sum_i |d_i| = n$, which may be preprocessed. In the *document listing* problem, we are given an online query comprising of a pattern string p of length m and our goal is to return the set of all documents that contain one or more copies of p. In the closely related *occurrence listing* problem, we output the set of all *positions* within the documents where pattern p occurs. In 1973, Weiner [24] presented an algorithm with $O(n)$ time and space preprocessing following which the occurrence listing problem can be solved in time $O(m + output)$ where $output$ is the number of positions where p occurs; this algorithm is clearly optimal. In contrast, no optimal algorithm is known for the closely related document listing problem, which is perhaps more natural and certainly well-motivated.

We provide the first known optimal algorithm for the document listing problem. More generally, we initiate the study of pattern matching problems that require retrieving documents matched by the patterns; this contrasts with pattern matching problems that have been studied more frequently, namely, those that involve retrieving all occurrences of patterns. We consider document retrieval problems that are motivated by online query processing in databases, Information Retrieval systems and Computational Biology. We present very efficient (optimal) algorithms for our document retrieval problems. Our approach for solving such problems involve performing "local" encodings whereby they are reduced to range query problems on geometric objects – points and lines – that have color. We present improved algorithms for these colored range query problems that arise in our reductions using the structural properties of strings. This approach is quite general and yields simple, efficient, implementable algorithms for all the document retrieval problems in this paper.

1 Introduction

We are given a collection D of text documents $d_1, \ldots, d_k$, with each d_i being a string, $\sum_i |d_i| = n$; this set may be preprocessed. In the *document listing* problem, online query comprises a pattern string p of length m and our goal is to return the set of all documents that contain one or more copies of p. In the *occurrence listing* problem which is closely related, we output the set of all positions in the different documents where pattern p occurs.

In 1973, Weiner [24] introduced the *suffix tree* data structure and proved that with $O(n)$ preprocessing time and space, each query can be answered in $O(m + occ(p))$ time, where $occ(p)$ is the number of positions where p occurs. Hence, this algorithm is *output sensitive*, and optimal for the occurrence listing problem. Suffix trees have since found numerous applications [11].

Weiner's algorithm does not solve the document listing problem optimally. This is because it uses the suffix tree data structure to index all suffixes of the documents so that when processing pattern p, it quickly finds a node such that the leaves in the subtree rooted at that node corresponds to all the occurrences of p in D. These leaves can be retrieved trivially in optimal time; however, in order to retrieve the distinct documents that contain p, we can not afford to scan all the occurrences of p in D since p may appear many times in any given document. Output size for the document listing problem is thus always smaller (and may, in the worst case be substantially smaller) than that for the occurrence listing problem. An optimal algorithm for the document listing algorithm has to be output sensitive. The best known algorithms for the document listing problem are suboptimal by at least a logarithmic factor (See 1.1 for the known bounds); furthermore, they are fairly sophisticated, using DAG edges added to the suffix tree and relying on fractional cascading. This contrasts with the simple, optimal solution Weiner provided for the occurrence listing problem.

This state of our knowledge is surprising. In many respects, the document listing problem seems to be far more natural than the occurrence listing problem from an application-specific point of view (say on the web or information retrieval systems where one first wants to retrieve documents that contain one or more patterns and only perhaps later determine the locations in those documents where the patterns appear). Nevertheless, the document listing problem seems to have received considerably less attention than the occurrence problem in Combinatorial Pattern Matching community. In fact, a more general phenomenon seems to be that off line problems that output pattern occurrences (e.g., find all occurrences of all patterns that repeat many times within a string, find all occurrences of all pairs of patterns that appear close together, etc) have been studied more extensively than online problems which output documents that contain patterns (e.g., find documents that contain many copies of a given pattern, find documents that contain the two given patterns appearing close together, etc). Con-

*AT&T Labs – Research, 180 Park Avenue, Florham Park, NJ 07932. Email: muthu@research.att.com

sequently, providing optimal algorithms for the document listing and other online document retrieval problems has remained open while related occurrence listing problems have been well studied.

In this paper, we present an optimal algorithm for the document listing problem. Our algorithm is quite simple, and easily implementable. We also initiate the study of online document retrieval problems in general: we present efficient algorithms for several such basic problems. In what follows, we will describe our problems and describe our results as well as techniques.

1.1 Document Retrieval Problems. We are given a *library* D of text documents $d_1, \ldots, d_k$, each d_i being a string over alphabet set Σ. We have $\sum_i |d_i| = n$ and $\max_i |d_i| \leq \ell$. The library may be preprocessed. Online queries differ from problem to problem.

Document Listing Problem. A basic problem is that of *document listing*. Here, the query `list(`p`)` consists of a pattern p of length m, and the goal is to return the set of all documents in the library in which p is present, that is, the output is $\{i \mid d_i[j \cdots j+m-1] = p \text{ for some } j\}$. Any full text search database [22] or information retrieval system [2] must solve the document listing problem. This problem has to be contrasted with the *Occurrence Listing Problem* where the online query is `occ(`p`)` and the goal is to output the set of all locations in the documents where p occurs, that is, the output is $\{(i,\ell) \mid d_\ell[i \cdots i+m-1] = p\}$. As mentioned above, the occurrence problem can be solved in optimal bounds [24]: $O(n)$ time and space preprocessing and $O(m + output)$ query processing time. In contrast, known algorithms for the document listing problem take time $\Omega(m \log n + output)$ for query processing with $O(n)$ preprocessing [20], or $\Omega(m + \log n + output)$ time with $\Omega(n \log n)$ preprocessing [7, 21]. These algorithms are sophisticated: they rely on augmentation of the classical suffix tree data structure with directed edges forming DAGs, together with fractional cascading. ∎

Document Mining Problem. The *document mining* problem is a generalization of the document listing problem. We have a prespecified threshold K. Here, query `mine(p)` consists of pattern p of length m, and the goal is to return the set of all documents in the library that contain p *at least* K times. Formally, the output is $\{i \mid \text{there exist at least } K\ j\text{'s such that } d_i[j \cdots j+m-1] = p\}$. This query is useful in data mining contexts where the goal is to determine the documents that contain the given pattern a significantly large number of times. Mining textual data has been studied in the database community (See references within [13, 23]), but mainly from an off line point of view where one finds all substrings that occur significantly often. Our online query context, which is quite natural, appears to be new. ∎

Repeats Problem. The *repeats* problem is as follows. We have a prespecified threshold K and the library. Online query `repeats(p)` consists of pattern p of length m and the goal is to return the set of all documents in the library that at least two occurrences of p close together, that is, separated by at most K positions. Formally, the output is $\{i \mid \text{there exist } j, l \text{ such that } j < l \leq j + K - 1 \text{ and } d_i[j \cdots j+m-1] = d_i[l \cdots l+m-1] = p\}$. Close repeats have evolutionary significance [11] in molecular data, and the problem of finding repeats has been studied extensively in Computational Biology [4]. The focus there has been on off line context, where given a string, the problem is to find all (maximal) occurrences of close substring repeats (See [5, 17] and references therein). Again, our online query version of the problem, intent on retrieving documents rather than all occurrences, appears to be new, but quite natural. ∎

Our discussion will focus on the three problems above, but several variations will be discussed in Section 7.

1.2 Our Results. Our main results are algorithms with $O(n)$ time and space preprocessing so that `list(`p`)`, `mine(`p`)` and `repeats(`p`)` can all be answered in time $O(m + output)$, where $output$ is the number of documents in the output for each case. Hence, our algorithms are optimal in both preprocessing and query processing.

For the mining and repeats problems, the case when threshold K is not prespecified, but rather varied as part of the query is of interest too. We present algorithms with near-linear preprocessing time and space, and optimal query time of $O(m + output)$ in that case.

All our algorithms work even when the strings are drawn from a large (polynomial sized) alphabet; they are quite simple, and easily implementable.

1.3 Technical Overview. Studying our document retrieval problems may appear to be a case of "Theory following Practice" because many information retrieval systems routinely solve the document listing problem, as do web search engines and full text databases. Furthermore, they are quite efficient; in contrast, no nontrivial (or optimal) theoretical algorithms are known for some of these problems. This is because systems usually see the query pattern p as a keyword and typically use inverted lists that list all documents containing a keyword. In our problems, query pattern p is an arbitrary substring and not necessarily a (key)word; this is more suitable on specific inputs such as molecular sequences where words are not natural lexicon units and in general where one may search for substrings of word combinations. For our case, inverted lists are inadequate. Also, explicitly storing the document lists for all queries will be highly space-inefficient. So our problems are challenging, and new theoretical ideas are needed to solve these problems

and obtain nontrivial or optimal bounds.

The document retrieval problems are simple to state. It is our experience that several approaches present themselves naturally for solving these problems, each entailing challenging data structural problems – we have stated some of them during the course of presenting our solutions (Problems $1-8$ in what follows). Unfortunately, optimal solutions are not known for these data structural problems. Hence additional algorithms are needed.

Our overall approach is as follows. We perform "local encoding" which consists of chaining each document suffix to another nearby based on variety of pairwise criteria. From this, we are able to generate range query problems to solve our document retrieval problems; the range query problems have objects drawn from different documents which may be thought of as different "colors". No optimal algorithms are known in general for the colored range query problems, but for the instances we get in our reductions, we are able to design optimal algorithms based on the structure of the queries. For variations wherein K is not a prespecified threshold, we identify and use considerable structure in the problem (see for example, Lemma 4.2) which we believe will be of independent interest. This overall approach is quite general and we expect it to find applications elsewhere.

1.4 Map. In Section 2, we describe the suffix tree and other data structural primitives we need. We discuss the document listing, mining and repeats problems in Sections 3, 4, 5 respectively. Concluding remarks are in Section 6. We can extend our results to variations of online document retrieval problems we have not discussed so far, but are nevertheless significant; see Section 7 for more details.

2 Data Structure Primitives

We will assume string documents are drawn from a fixed size alphabet (and consider other alphabet models in Section 7). Given a string document s, its suffix tree $T(s)$, is a compressed trie of all the suffixes of s. Each edge is labeled with a substring of s, as a $O(1)$ space representation using pointers into s. For any node v, we let σ_v denote the string obtained by concatenating the substrings labeling the edges on the path from the root to v in the order they appear. Each leaf l is in an one-to-one relationship with the suffixes of s, that is, $\sigma_l = s[j \ldots |s|]$ for a unique j. At each node in the trie, children are sorted based on the first symbol on the strings labeling the edges (by definition, these symbols are unique). The definition of a suffix tree extends naturally to multiple string documents, and it is called the *generalized suffix tree* in the literature [11]. A detail is that a leaf in the generalized suffix tree may represent (identical) suffixes in many documents: we assume that each such leaf has dummy leaf children, one for each such suffix, so that each leaf now represents a unique suffix in one of the documents.

Generalized suffix tree is used to index the suffixes and search for patterns. Let p be any pattern; we define its *locus* u_p as the node in the generalized suffix tree such that σ_{u_p} has the prefix p and $|\sigma_{u_p}|$ is the smallest of all such nodes u_p. Note that the locus of p does not exist if p is not a substring of one of the documents. Therefore, given p, finding its u_p suffices to determine if p occurs in any of the documents. Weiner [24] showed that generalized suffix tree of the library of documents can be built in $O(n)$ time and space so that for any pattern p with $|p| = m$, in $O(m)$ time we can determine if its locus exists, and its locus in case it exists.

For any two nodes u and v in (suffix) tree T, *lca(u,v)* denotes their lowest common ancestor. A given tree of size n can be preprocessed in $O(n)$ time so that *lca(u,v)* can be determined in $O(1)$ time for any u, v as shown in [14], a fact we use.

3 The Document Listing Problem

In this section, we describe our optimal algorithm for the document listing problem.

3.1 Optimal Algorithm for the Document Listing Problem. Our preprocessing is as follows. We build the generalized suffix tree for the library of documents. Perform an inorder traversal of the tree and label all the leaves $l_1, \cdots, l_n$ in the order in which they appear; the leaves correspond to the n suffixes in the library. Note that $i < j$ implies $\sigma(l_i)$ is lexicographically smaller than $\sigma(l_j)$.

Define D to be the array of documents the leaves correspond to in order, and L be the array of the occurrences of the leaves in order. More formally, $D[i] = j$ and $L[i] = k$ if $\sigma(l_i)$ represents the suffix $d_j(k \cdots |d_j|)$. Our algorithm proceeds as follows.

Step 1. Given p, find its locus u_p. Recall that if p is not present as a substring in any of the documents, its locus does not exist. Henceforth, we focus on the case when u_p exists.

Step 2. From u_p, we find l_s and l_b defined as follows. Let l_s and l_b be the leftmost and rightmost leaves that are the descendants of u_p, respectively. That is, $\sigma(l_s)$ starts with $\sigma(u_p)$ (and hence p) and is lexicographically the smallest suffix in the library with this property. Likewise for $\sigma(l_b)$ which is lexicographically the biggest suffix in the library that starts with $\sigma(u_p)$. Observe that all suffixes in the library that start with p are larger than $\sigma(l_s)$ and smaller than $\sigma(l_b)$ in the lexicographic ordering. Hence, the following observation is immediate: *a suffix in the library begins with p if and only if it is represented by one of the leaves $l_s, \ldots, l_b$.*

Problem 1. *(Colored Range Query Problem)* Given an array $A[1 \cdots n]$ of colors to be preprocessed, online query consists of an interval $[i, j]$, and the goal is to return the set of distinct colors that appear in $A[i, j]$. ∎

Our document listing problem can be reduced to the col-

ored range query problem as follows. Consider array D and let each document ID be its color. Then the document listing problem with query p reduces to finding the distinct colors that occur in $D[s, b]$ because of the observation above. The best known algorithm for the colored range query problem [16] will solve this problem in time $O(\log n + output)$. However, this will only provide an $O(m + \log n + output)$ time algorithm for the `list(`p`)` query which is not optimal, in particular, when $|p| = o(\log n)$ and $output = o(\log n)$.

We proceed to solve this problem optimally by "locally" encoding array D suitably. We define a new array C as follows based on D. Informally, C chains the occurrences of suffixes from a given document in the lexicographic order. This is essentially stored in C as a linked list of pointers from each suffix of a given document to its "predecessor" in the document, i.e., the largest of the lexicographically smaller suffixes contained in that document. Formally, we set $C[i] = j$ if $j < i$, $D[i] = D[j] = k$, and j is the largest index with this property. (If no such j exists, we set $C[i] = -1$, a boundary value.) Now we make a crucial observation about our encoding C.

LEMMA 3.1. *Document i contains p if and only if there exists precisely one $k \in [s, b]$ such that $D[k] = i$ and $C[k] < s$.*

Proof. Consider the case when document i contains p. Let $j_1, \ldots, j_q$ be positions in it that contain p. Let $l_{k_1}, \ldots, l_{k_q}$ be the leaves that represent the suffixes $d_i[j_1 \cdots |d_i|], d_i[j_2 \cdots |d_i|], \cdots, d_i[j_q \cdots |d_i|]$ in lexicographically sorted order, that is, $k_1 < k_2 < \cdots < k_q$. Clearly, $k_i \in [s, b]$ for all i. Let $k = \min_i k_i$. Trivially, $D[k] = i$. Also, by definition, $C[k] = \alpha$ if $\alpha < k$ and $D[\alpha] = i$. In that case, $\sigma(l_\alpha)$ is in i and is lexicographically smaller than $\sigma(l_k)$. By definition of k, it follows that $\sigma(l_\alpha)$ does not start with p. Hence, it follows that $\alpha < s$. Furthermore, note that, for $k_i \neq k$, $C[k_i] \geq k$ by definition and hence, $C[k_i] > s$; therefore, there is a unique value of k that satisfies the lemma for a given document i. The other direction is straightforward. ∎

Step 3. Given s and b, find all $i \in [s, b]$ such that $C[i] < s$ and output $D[i]$. From the lemma above, it follows that the documents that contain p are all *uniquely* listed in the output.

This step uses a simple divide and conquer procedure with the following data structural primitive.

Range Minima problem. We are given an array $A[1 \cdots n]$ to be preprocessed. A Range Minima Query (RMQ) specifies an interval I and the goal is to find $\min_{i \in I} A[i]$. This problem can be solved in $O(n)$ time preprocessing and $O(1)$ time per query [9]. This algorithm can be easily modified to output the i for which $A[i]$ achieves the minimum for the query I in same bounds. ∎

We will use the RMQ to solve our problem. In order to find all $i \in [s, b]$ with $C[i] < s$, we find $j \in [s, b]$ such that $C[j]$ is the smallest in $[s, b]$ using the RMQ query with $[s, b]$. If $C[j] > s$, then output is empty and we stop. Else, we output $D[j]$ and solve the same problem on $[s, j-1]$ (find all $i \in [s, j-1]$ with $C[i] < s$) and $[j+1, b]$ (find all $i \in [j+1, b]$ with $C[i] < s$). This procedure clearly outputs each of the $i \in [s, b]$ with $C[i] < s$; furthermore, no such i is output more than once because of the nature of the divide and conquer. We conclude,

THEOREM 3.1. *There exists an $O(n)$ time algorithm to preprocess a library of documents of total size n such that* `list(`p`)` *query can be solved in time $O(m + output)$ where "$output$" is the number of documents in that contain p.*

Proof. Correctness follows from the algorithm description. Preprocessing comprises: (1) constructing the generalized suffix tree and determining l_s and l_b for each of the nodes in the generalized suffix tree, and (2) constructing arrays D, L and C. Each of these takes $O(n)$ time. Steps 1 and 2 can be done in $O(m)$ time. Step 3 takes time $O(output)$ using the time bound for RMQ, and observing that the RMQ is not applied once the entries in the range no longer satisfy the output criterion. ∎

Our solution above in fact solves the colored range query problem in optimal bounds $O(output)$ time per query (and improves [16]); we will use this result in the remainder of the paper.

4 Document Mining Problem

We will present our optimal algorithm for the document mining problem when the threshold K is fixed, and later, show an efficient algorithm when K varies as specified in the query.

4.1 Fixed K Case.

First, we will introduce two, fairly natural, data structural problems that may be of independent interest. Later, we will show how our document mining problem reduces to them.

Problem 2. We are given an array $A[1 \ldots n]$ of nonnegative integers which we may preprocess, and a threshold integer K. Each query specifies a range $[a, b]$, and the goal is to return the numbers which appear at least K times in $A[a \cdots b]$. More precisely, the output is

$$\{j \mid A[i_1] = A[i_2] = \cdots = A[i_k] = j \ \&\ i_l \in [a, b] \ \&\ k \geq K\}$$

Our goal is to find all such j's in total time proportional to the output size, with linear preprocessing time and space. Despite extensive results on various range searching problems [1, 16], we are not aware of any result that directly addresses this problem or meets this goal. ∎

Problem 3. *(Colored Interval Containment Problem)* We are given S, a set of n colored intervals, each interval with a color drawn from $1 \cdots n$, which may be preprocessed. Each query comprises an interval I and the goal is to output the set of colors such that for each color, there is some interval in S of that color that is completely contained in I. Again, the goal is to find all such colors in total time proportional to the output size, with linear preprocessing time and space.

This is a generalization of a geometric intersection problem to objects with colors. Many such problems have been studied with colored objects [16, 1]. Nevertheless, we do not know of a result that directly applies to this problem. For example, in [16], a colored interval *intersection* problem has been studied where given a query I, the goal there is to return the colors such that for each color, there is some interval in S of that color that *overlaps* I (i.e., not necessarily contained in I); this does not solve our intersection problem. The colored interval containment problem can be reduced to a two dimensional range searching problem with colored points as follows. Replace interval $[l, r]$ of color c with a point (l, r) of color c for each interval, and query $[a, b]$ by a rectangle query $[a, b] \times [a, b]$ that returns distinct colors on the points in the query rectangle. The best known bounds for this problem takes $O(n \log n)$ space and $O(\log^2 n + output)$ time per query or $O(n \log^2 n)$ space and $O(\log n + output)$ time per query [16], neither of which meets our goal. ∎

Now we will show first how these problems capture the inherent difficulty in our document mining problem by reducing the mining problem to these problems. As in the case of the document listing problem, we will construct the generalized suffix tree of the library, consider its leaves in order and determine L and D arrays. We will also perform Steps 1 and 2 as before and determine l_s and l_b for the given p.

One way to solve our document mining problem is to consider the D array, and determine i that appear at least K times in $[s, b]$. Clearly this will solve our mining problem since occurrence of p in a document has a unique entry for that document in $D[s \cdots b]$. In order to solve the document mining problem optimally, we must return all such documents in time proportional to the output size. This is precisely problem 2.

Another way to solve our document mining problem is to construct a "local encoding" encoding as we did for the document listing problem. More precisely, we define an array E as follows. Informally, E stores pointers from each location to denote the extent of *precisely* K occurrences of the same document to its left. Formally, $E[i] = j$ if and only if for $j < i$, $D[j] = D[i]$ and there are precisely K occurrences of $D[i]$ in $D[j \cdots i]$. This may be thought of as an interval $[j, i]$ with color $D[i]$, which gives the input to problem 3. It is now clear that a document l contains pattern p at least K times if and only there is an interval of color l that is completely contained in $[s, b]$. (Note that if a document contains $K + 1$ or more occurrences of p, two or more intervals of same color will be contained in $[s, b]$ due to this reduction.) Hence, solving the colored interval containment problem (problem 3) will solve the document mining problem.

In what follows, we will solve the document mining problem optimally. Our solution in fact solves problems 2 and 3 optimally for the special instances that arise in the reduction above, but not in general. This is because one of the crucial structural properties that helps us solve the document mining problem is that the set of queries is only linear in size and that they are hierarchical (the queries are nodes of the tree which corresponds to a set of intervals with the property that if two intervals overlap, one contains the other); this is not true for the queries in Problems 2 and 3 above in general. Our algorithm has the following steps.

1. We consider the generalized suffix tree of the library and color certain nodes so that the document mining problem reduces to one of finding distinct colors on descendants of nodes in a tree.

2. We linearize the tree and reduce the colored descendants problem to colored range query problem which we solve as in the previous section optimally.

In Step 1, we start by constructing the generalized suffix tree of the library. The crux of our solution relies on "coloring" some of the nodes in the generalized suffix tree which we call *mine coloring* described below. Let l_i's represent the set of leafs in lexicographic order as before. Consider any leaf l_i. Determine i^*, $i^* \leq i$, such that l_i and l_{i^*} belong to the same document and there are precisely K leaves between (and including) l_{i^*} and l_i that are from the same document as l_i and l_{i^*}. Note that l_{i^*} may not exist for some l_i (this holds for the $K - 1$ lexicographically smallest suffixes in each document). Let $lca(u, v)$ denote the least common ancestor of nodes u and v in the tree. We color $lca(l_i, l_{i^*})$ with color d, if l_i is from document d. Note that a node in the generalized suffix tree may get zero, one or more colors. The coloring thus obtained is called the *mine coloring*. Mine coloring has the following crucial property.

LEMMA 4.1. *For any node v in the generalized suffix tree, document d contains σ_v at least K times if and only if there is a node colored d in the subtree rooted at v.*

Hence, our mining problem reduces to the following problem.

Problem 4. *(Colored Descendants Problem)* We are given a tree of size n in which each node u is colored with a set s_u of colors, $s_u \subseteq \{1, \ldots, k\}$. This tree may be preprocessed. Given a node v as the query, return the set of colors S_v such

that each color in S_v is found on one of the colors associated with the descendants of v. Formally, the goal is to return S_v where each $c \in S_v$ satisfies $c \in s_u$ for some u descendant of v. ∎

In our reduction, in fact, $\sum_u |s_u| = \sum_i |d_i| = O(n)$ because every leaf generates at most one color at some node.

In Step 2, we solve the colored descendants problem on trees. We do a preorder traversal of the tree arrange the nodes in an array N in the order in which they are visited. There is a technical detail: each node v is entered consecutive $|s_v|$ times in N, each time with a distinct element from s_v (this in particular means that a node v with empty s_v will not be represented in N). Thus each $N[i]$ is a color associated with one of the nodes in the tree, the colors associated with a given node are consecutive, and furthermore, the colors associated with different nodes appear in the order of traversal of the nodes.

Given the construction of N array above, we can reduce the document mining problem to a colored range query problem. Let F_v denote the smallest index in N that represents s_v for any v and L_v denote the largest index in N that represents c_u for any descendant of u. Observe that by the way we ordered N, all the descendants of node v will be found in $N[F_v, L_v]$ which is useful. Given query v, it is now easy to observe that outputing S_v is same as outputting the distinct colors in N in the range $[F_v, L_v]$. That completes the reduction. We solve the colored range query problem on N as we did in the previous section, optimally. That lets us conclude,

THEOREM 4.1. *There exists an $O(n)$ time and space algorithm to preprocess the library so that we can answer any given* `mine(`p`)` *query in time $O(m + output)$, where "$output$" is the number of documents that satisfy the query.*

4.2 Variable K. In the previous section, we let K be fixed, independent of the input. In what follows, we provide another algorithm that is efficient for any value of K specified as part of the query. Formally, the query is now `mine(`p`,`K`)`, that is, find all documents that contain p at least K times, where p and K are now query parameters. The algorithm here relies on identifying certain structure in mine coloring, and sparsifying the coloring.

We make a structural observation about mine coloring. Let l_i^q, $i = 1, \ldots, |d_q|$, be the leaves corresponding to suffixes of document q in lexicographically sorted order smallest to largest, that is, left to right in the generalized suffix tree.

LEMMA 4.2. *Any node v in the generalized suffix tree in which $q \in s_v$ in the mine coloring lies on the path from the root to leaf l_{Ki}^q for any integer i.*

Proof. Focus on leaves $l_{K(i-1)}^q \cdots l_{Ki}^q$. Mine coloring includes coloring nodes $lca(l_{K(i-1)}^q, l_{Ki}^q)$, $lca(l_{K(i-1)+1}^q, l_{Ki+1}^q)$, $lca(l_{K(i-1)+2}^q, l_{Ki+2}^q)$, $\ldots lca(l_{K(i-1)+K-1}^q, l_{Ki+K-1}^q)$ with color q. Observe that for leaves a, b, c, $lca(a, c) = lca(lca(a, b), lca(b, c))$ if b lies between a and c in the preorder traversal of leaves. Hence, for any j, $lca(l_{K(i-1)+j}^q, l_{Ki+j}^q) = lca(lca(l_{K(i-1)+j}^q, l_{Ki}^q), lca(l_{Ki}^q, l_{Ki+j}^q))$. Since $lca(a, b)$ is a node that is on the path from the root to a and to b, it follows that all nodes marked q due to leaves $l_{K(i-1)}^q \cdots l_{Ki}^q$ lies on the path from the root to l_{Ki}^q. ∎

The lemma above shows very useful structure: the nodes colored q are placed in $|d_q|/K$ root-to-leaf paths. This helps us filter some colors out of the mine coloring. In particular, for each such path, we choose the representative to be the node of largest depth that is colored q; we remove color q from the coloring of *other* nodes on the path. We refer to this as *sparse mine coloring*. As before, now we reduce the document mining problem to the colored descendants problem, but with sparse mine coloring. The crux is that sparse mine coloring suffices for the correctness as shown below (proof omitted).

LEMMA 4.3. *For each node v, let S_v be the output with mine coloring and S'_v be that with sparse mine coloring. We have $S_v = S'_v$.*

The progress we have made in going from general mine to sparse mine coloring is in reducing the problem size. In sparse mine coloring, we have $\sum_v |S'_v| = \sum_q |d_q|/K$ sized sets altogether which is smaller than $\sum_v |S_v| = \sum_q |d_q|$ for any K; we will exploit this fact shortly.

We will now show how to compute the representative for each path efficiently. It is trivial to do so in time $O(|d_q|)$ per document q, however, here we will show an algorithm that requires $O(n)$ preprocessing altogether, following which, the representatives can be chosen significantly faster for different values of K. We will show an algorithm to determine the representative for the root-to-leaf path to l_{Ki}^q. Focus on leaves $l_{K(i-1)}^q \cdots l_{Ki}^q$. Recall from above that mine coloring involves coloring nodes $lca(lca(l_{K(i-1)+j}^q, l_{Ki}^q), lca(l_{Ki}^q, l_{Ki+j}^q))$ for $j = 0, \ldots, K$ with color q, and that the representative we seek is the largest depth node in this collection. Define $depth(v)$ to be the depth of node v. We have (proof omitted),

LEMMA 4.4. $depth(lca(l_{K(i-1)+j}^q, l_{Ki}^q)) \geq depth(lca(l_{K(i-1)+j+1}^q, l_{Ki}^q))$
$depth(lca(l_{Ki}^q, l_{Ki+j}^q)) \leq depth(lca(l_{Ki}^q, l_{Ki+j+1}^q))$.

We need to find j such that $depth(lca(lca(l_{K(i-1)+j}^q, l_{Ki}^q), lca(l_{Ki}^q, l_{Ki+j}^q)))$ is smallest. From the previous lemma, it follows that this function has a minimum value, and it suffices to do a (careful) binary search with the lca oracle to determine that minimum. We

omit the details. This takes time $O(\log K)$ per path, and in all $O(\sum_q \frac{|d_q| \log K}{K})$ over all documents.

THEOREM 4.2. *There exists an $O(n)$ time and space algorithm to preprocess the library so that with further preprocessing of $O(\sum_q \frac{|d_q| \log K}{K}) = O(\frac{n \log K}{K})$ time and $O(\frac{\sum_q |d_q|}{K}) = O(\frac{n}{K})$ space for any K, we can answer any given* `mine(`p, K`)` *query in time $O(m + output)$, where "$output$" is the number of documents that satisfy the query.*

If we apply the algorithm to the case of κ chosen values of K, where $\kappa = \{1, \lceil(1+\varepsilon)\rceil, \lceil(1+\varepsilon)^2\rceil, ...\}$, for some constant ε, we get an algorithm with $\sum_{K \in \kappa} O(\frac{n \log K}{K}) = O(n)$ preprocessing time and space which is optimal, and query time is optimal too. If one were to build a data structure for *all* values of K, we can indeed do so by applying the theorem above, and the preprocessing running time becomes $O(n + \sum_{K=1,...,\ell} \frac{n \log K}{K}) = O(n \log^2 \ell)$, and space becomes $O(n + \sum_{K=1,...,\ell} \frac{n}{K}) = O(n \log \ell)$, which is still very efficient for preprocessing; query time remains optimal.

5 Document Repeats Problem

We first present an optimal algorithm for the document repeats problem with fixed K. Then we design an efficient algorithm for K varying with queries.

5.1 Optimal Algorithm for Fixed K. Consider:

Problem 5. *(Colored Gaps Problem)* We are given an array $A[1 \ldots n]$ of nonnegative integers, each with a color, and a threshold K. We may preprocess this information. Each query specifies a range $[a, b]$, and the goal is to return the distinct colors such that at least two integers of that color in $A[a, b]$ are less than K apart. More precisely, $\{c \mid$ for some $i, j \in [a, b]$, $A[i]$ & $A[j]$ are colored c, $|A[i] - A[j]| \leq K\}$. Our goal is to find all such j's in total time proportional to the output size, with linear preprocessing time and space. We are not aware of any previous result that addresses this problem or meets this goal. ∎

We can reduce the document repeats problem to Problem 5; we omit the reduction here since it does not help us solve the document repeats problem optimally. In what follows, we present an algorithm to solve the document repeats problem optimally. As in Section 4, we reduce our problem to the colored descendants problem in a tree. What will differ is the method for coloring tree nodes. We begin by constructing the generalized suffix tree of the library. Consider any index i in L. Determine i^*, $i^* \leq i$, such that $L[i]$ and $L[i^*]$ belong to the same document, $|L[i^*] - L[i]| \leq K$, and i^* is the largest index with this property. Note that i^* may not exist for some l_i (eg., for the leftmost $K-1$ suffixes in each document). We color $lca(l_i, l_{i^*})$ with color q, if l_i is from document q. As before, a node in the generalized suffix tree may get zero, one or more colors. The coloring thus obtained is called the *repeats coloring*. It has the following crucial property, which, using Section 4, gives theorem below.

LEMMA 5.1. *For any node v in the generalized suffix tree, document q contains σ_v repeated with at most K positions between them if and only if there is a node colored q in the subtree rooted at v.*

THEOREM 5.1. *There is a $O(n)$ time and space algorithm to preprocess the library so that any* `repeat(`p`)` *problem can be solved in time $O(m + output)$ where "$output$" is the number of documents that satisfy the query.*

5.2 Varying K. Now let threshold vary with the query, that is, the query is `repeats(`p, K`)`. In the document mining problem, we found structure in coloring of nodes that helped us sparsify the coloring and obtain efficient algorithms for general K. This approach does not work since the structural observations there provably do not hold for document repeats problem (eg., Lemma 4.2). Therefore, one needs a different approach. Our solution will first get an algorithm with slightly suboptimal query processing time by reducing the problem to the following; later, the algorithm will be improved to obtain optimal query processing time.

Problem 6. *(Colored Segment Intersection Problem)* We are given n colored vertical segments to be preprocessed. Each query is a horizontal segment, and the goal is to return the distinct colors on the vertical segments that intersect the query. ∎

Our reduction has a few steps, and proceeds as follows. We consider suffix trees $T_1, \ldots, T_k$ of the k documents respectively as well as their generalized suffix tree constructed jointly for all the documents and denoted T. On each T_i, we label the leaves l_i^j to denote the suffix j of document d_i. At each node $v \in T_i$, we attach a label denoted $M(v)$ where $M(v) = \min\{|j_1 - j_2| \mid l_i^{j_1}$ & $l_i^{j_2}$ are descendants of $v\}$. By the properties of suffix trees, observe that for each node v in one of the T_i's, there is a node $f_i(v)$ in T such that $\sigma_{f_i(u)} = \sigma_v$. We *mark* each node u in the generalized suffix tree with $M(v)$ with color i for all v such that $f_i(v) = u$. Note that each u gets a set of such colored marks (but no more than one of a color). Now we linearize the generalized suffix tree and construct an array N of the colored marks: we do a preorder traversal of the tree and arrange the colored marks on nodes in an array N in the order in which they are visited. There is a technical detail: each node v is entered once for each colored mark on it (this in particular means that a node v without any colored marks will not be represented in N). Thus each $N[i]$ is a colored mark associated with one of the nodes in the tree, the colored marks associated with a given node are consecutive, and furthermore, the

colored marks associated with different nodes appear in the order of traversal of the nodes.

Given the construction of N array above, we can reduce the document repeats problem to colored segment intersection problem as follows. Each $N[i]$ is replaced by a vertical segment starting from $(i, N[i])$ and ending at $(i, n+1)$ (since $N[i]$ represents a position in one of the documents, $N[i] \leq n$) with the color of $N[i]$. Given a pattern p and K as part of the query, we find u_p in T as in the document listing problem in Section 3. Let F_v denote the smallest index in N that represents a colored mark of v and L_v denote the largest index in N that represents a colored mark of any descendant of u. Observe as before that by the way we ordered N, all the colored marks on descendants of node v will be found in $N[F_v, L_v]$. Our query now can be written as the horizontal segment starting at (F_v, K) and ending at (L_v, K). We claim that the distinct colored segments that intersect this query yields the distinct documents that satisfy the query `repeat(`p`,`K`)`; given our description above, it is not hard to prove this claim. That completes the reduction.

Marking $M(v)$'s for all nodes of all T_i's takes time $O(n \log n)$ using the results in [3]. The colored segment intersection problem can be solved with $O(n \log n)$ preprocessing time and space so that each query can be answered in time $O(\log n + output)$ where $output$ is the number of colors in the output [16]. Using these two results with our reduction above to solve the document repeats problem would give an algorithm that takes $O(n \log n)$ time and space for preprocessing with $O(m + \log n + output)$ time for each query, where $output$ is the number of documents that satisfy the repeats query. Hence, the query time is not optimal in pattern size when m and $output$ are $o(\log n)$ in general.

In order to obtain an algorithm that has optimal query time, we *amortize* the work of reading the pattern against the query time. This can be done trivially when pattern is large enough, that is $m = \Omega(\log n)$. When m is small, i.e., $o(\log n)$, we adopt another approach. The natural approach will be to consider all patterns of small size as candidates, and precompute the solution for each. This is what is known as the "Four Russians" trick. but it is difficult apply directly when (a) alphabet size is large and (b) solution set we store for each candidate is large, as in our problem. Instead, we store a preprocessed "solution" set for those patterns of small length that actually appear as nodes in the generalized suffix tree and not for all patterns of small length; furthermore, the solution set is stored in appropriate order so that it supports queries with different values of K. Formally, we define the *string depth* of node v to be $|\sigma_v|$. For each node v in the generalized suffix tree of string depth at most $\log n$, we store the marks in the subtree rooted at v in the portion $[F_v, L_v]$ of the linearized array N; we keep these marks in a sorted order, smallest to largest. For each color, it suffices to keep the smallest value of that color. Observe that for any ℓ, the *total* set size we have stored over all nodes of string depth precisely ℓ is at most n. Hence, the total space and preprocessing time so far is $O(n \log n)$ if we execute the sorting operation in linear time by bucket sorting. Given pattern p, we find u_p as in our algorithm description by walking down the tree as usual. We scan the marks stored with node u_p in the increasing order of values until the mark value exceeds K, and output the colors encountered. Query processing therefore takes time $O(m + output)$.

Combining the algorithm above for $m \leq \log n$ with the algorithm described earlier using colored segment intersection for $m > \log n$, we have:

THEOREM 5.2. *There exists an $O(n \log n)$ time and space algorithm for the document repeats problem so that any query* `repeats(`p`,`K`)` *can be answered in $O(m + output)$ time, where "$output$" is the number of documents that satisfy the query.*

6 Concluding Remarks

We have presented the first known optimal algorithm for the document listing problem, while, curiously, the closely related (perhaps less natural) problem of occurrence listing was solved optimally in 1973 by Weiner [24]. We have initiated the study of text matching problems that retrieve documents rather than occurrences, and presented efficient algorithms for several such problems motivated from text data mining, computational biology and indexing full text databases. Our algorithms are simple and implementable, and we expect them to be of use in practice. We hope novel document retrieval problems get formulated and solved over time.

Our technique for solving the document retrieval problems relied on "local reductions" where we chained suffixes together based on pairwise constraints. This technique is quite powerful and we used it for all the problems we solved here. However, this technique will not suffice to solve document retrieval problem that have certain non-local constraints, for example, find all documents in which *all* occurrences of p followed by q are at least K positions apart. We leave open such problems with a large "certificate" for testing the desired property. Also, we believe problems $1-8$ are of independent interest: finding improved algorithms where possible, or applications for them would be of interest.

References

[1] P. Agarwal and M. van Kreveld. Polygon and connected component intersection searching *Algorithmica*, 1996, 15, 626-660.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley Longman Publ., 1999.

[3] A. Ben-Amram, O. Berkman, C. Iliopoulos, K. Park. The Subtree Max Gap Problem with Application to Parallel String

Covering. Proc. Symp on Discrete Algorithms (SODA), 1994, 501-510.
[4] G. Benson and M. Waterman A Method for Fast Database Search for All k-nucleotide Repeats. *Nucleic Acids Research*, Vol 22, No. 22, 1994.
[5] G. Brodal, R. Lyngso, Ch. Pedersen and J. Stoye. Finding maximal pairs with bounded gap. *Proc. 10th Combinatorial Pattern Matching* (CPM), Vol 1645, LNCS, 1999.
[6] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. *Proc. Foundations of Compter Science* (FOCS), 1997, 137-143.
[7] P. Ferragina. Personal Communication, 1999.
[8] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional Substring Indexing. *Proc. Principles of Database Systems* (PODS), 2001.
[9] H. Gabow, J. Bentley, R. Tarjan. Scaling and Related Techniques for Geometry Problems. *Proc. Symp Theory of Computing* (STOC), 1984, 135-143.
[10] P. Gupta, R. Janardan, M. Smid. Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization. *J. Algorithms* 1995, 19(2), 282-317.
[11] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge Univ Pr; ISBN: 0521585198 ; Dimensions (in inches): 1.44 x 10.30 x 7.41.
[12] T. Hagerup, P. Bro Miltersen and R. Pagh. Deterministic Dictionaries. To appear in *Journal of Algorithms*.
[13] J. Han, L. Lakshmanan and J. Pei. Scalable Frequent-Pattern Mining Methods: An Overview. *7th ACM Conf on Knowledge Discovery and Data Mining* (KDD), Tutorial, 2001.
[14] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338-355, May 1984.
[15] L. Hui. Color Set Size Problem with Application to String Matching. *Proc. Combinatorial Pattern Matching* (CPM), 1992, 230-243.
[16] R. Janardan and M. Lopez. Generalized intersection searching problems. *Intl J Comput. Geom. Applications*, 3(1993), 39-69.
[17] R. Kolpakov and G. Kucherov. Finding repeats with fixed gap. Technical Report RR-3901, INRIA, March 2000.
[18] U. Manber, G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *Proc. Symp on Discrete Algorithms* (SODA), 1990, 319-327.
[19] U. Manber and R. Baeza-Yates. An Algorithm for String Matching with a Sequence of don't Cares. *Information Processing Letters* (IPL), 1991, 37(3): 133-136.
[20] Y. Matias, S. Muthukrishnan, S. Sahinalp and J. Ziv. Augmenting Suffix Trees, with Applications. *Proc. European Symp on Algorithms* (ESA), 1998, 67-78.
[21] R. Hariharan. Personal communication, 1999.
[22] SQL full text indexing. http://larr.unm.edu/ owen/ SQLBOL70/html/$ca - co_1 5.htm$.
[23] Jaak Vilo. Discovering Frequent Patterns from Strings Technical Report, Department of Computer Science P.O.Box 26 FIN-00014. Check at http://citeseer.nj.nec.com/103438.html.
[24] P. Weiner. Linear Pattern Matching Algorithms. *Proc. 14th IEEE Symp on Switching and Automata Theory*, 1973, 1-11.

7 APPENDIX: Extensions

In this section, we mention extensions of our results and techniques.

7.1 Variants of Document Retrieval Problems. Many natural variants of document retrieval problems can be solved using the techniques we have presented in the paper. As an example, consider a problem closely related to the document listing problem, where the query is `list(NOT` p`)`, that is, list the documents that *do not* contain pattern p. In general, in Information Retrieval area, it is believed to be not a good practice to look for documents that do not contain a key word or phrase since such queries are often not very selective. Nevertheless, such queries may be useful in filtering a partially culled set of documents or in specialized data collections such as patent or biological databases. We present an optimal algorithm for this problem. Consider:

Problem 7. *(Complement Colored Range Query Problem)* We are given an array $A[1 \cdots n]$ of integer colors to be preprocessed. Each query is a range $[a, b]$ and the goal is to list the distinct colors that are *not* found in $A[a \cdots b]$. ■

In order to support the `list(NOT` p`)` query, we preprocess the library as before and build a generalized suffix tree, and find l_s and l_b for p as in Step 2 in Section 3. Now it is clear that solving the complement colored range query problem on D with range query $[s, b]$ suffices to support the `list(NOT` p`)` query. Although colored range query problem has been studied in the literature before in its many variations [16, 1], this complement version has not been studied.

We solve the complement colored range query problem using a variation of our "local encoding" trick. We append an instance of each color to the array. More formally, we let $A[n + i]$ to be the ith distinct color in $A[1 \cdots n]$, in some order. This technical step simplifies the algorithm description. So the combined array is of size $A[1 \cdots n + C]$ where C is the total number of distince colors in $A[1 \cdots n]$. Next, we do predecessor encoding as before, that is, we set $C[i] = j$ if $j < i$, $A[i] = A[j] = k$, and j is the largest index with this property. (If no such j exists, we set $C[i] = -1$, a boundary value.) Now we claim:

LEMMA 7.1. *Color i does not appear in $A[a, b]$ if and only if there exists precisely one $k \in [b + 1, n + C]$ such that $A[k] = i$ and $C[k] < a$.*

Therefore, solving the complement colored range query problem reduces to finding the integers in $C[b + 1, n + C]$ that are less than a, and returning the associated document identifiers. This can be solved in $O(output)$ time using RMQ as in Section 3.

THEOREM 7.1. *A library of documents of size n can be processed in $O(n)$ time such that* `list(NOT` p`)` *query can*

be solved in time $O(m + output)$ where "$output$" is the number of documents that do not contain p.

Similarly, we can solve the document mining problem where the goal is to retrieve documents that contain the query p at *most* K times, or the document repeats problem where the goal is to retrieve documents that contain the query p separated by at *least* K positions.

7.2 Two (or more) Patterns. All our document retrieval problems can be extended to multiple patterns. Since there are outstanding issues even with two patterns, we focus only on that case. The query `list(`p`,`q`)` – list all documents that contain all documents that contain *both* p and q – is of great interest in the database context [8]. The fundamental challenge here is to use subquadatic (in library size) preprocessing time and space, while taking time proportional to $|p|$ and $|q|$, and only *sublinear* in the library size in the worst case. Only such result known for this problem is in [8] where an $\tilde{O}(n^{3/2})$ time and space preprocessing algorithm is given which supports list queries in time $O(|p| + |q| + \sqrt{n} + output)$.

We can adopt and extend the approach in [8] to the other document listing problems. In what follows, we will present our result for the *document proximity problem*: `near(`p`,`q`)` query returns documents that contain p and q separated by at most K positions. The query can be solved by reducing it to the following problem:

Problem 8. *(Close Common Colors Problem)* Given an array A of n colored integers to be preprocessed, each query consists of two non-overlapping intervals I and J and the goal is to return the colors c such that there are an integer $i \in A[I]$ and an integer $j \in A[J]$ both colored c with $|j - i| \leq K$. ∎

We briefly describe how we solve this problem. We divide A into $\sqrt{n}$ disjoint segments of size $\sqrt{n}$ each. For each pair of segments s and t, we put a point (s, t) on the plane, colored c, for each c, such that there is an integer i in s and an integer j in t, both colored c with $|j - i| \leq K$. Any query I and J can be solved as follows. We break I (and J) into at most 3 parts, the leftmost and rightmost of size at most $\sqrt{n}$, and the middle comprising segments of size a multiple of $\sqrt{n}$, and label them I_l, I_r, I_m (J_l, J_r, J_m) respectively. For each integer in I_l, I_r (J_l, J_r), we can quickly determine if there is an integer of same color anywhere in J (I respectively), and differing by at most K, using an appropriate range query. It remains to check the integers in I_m and J_m for which we pose a colored range query on the two dimensional points we have preprocessed. This reduction is different from [8] (in fact, we do not know how to directly use their reduction technique here) and is considerably simpler.

THEOREM 7.2. *There exists an $O(n^{3/2} \log n)$ time and space preprocessing of an $O(n)$ sized library that supports* `near(`p`,`q`)` *queries in time $O(|p|+|q|+\sqrt{n}\log n+output)$ where "$output$" is the number of documents that satisfy the query.*

The `near` query is part of the SQLServer (Microsoft database system) query language [22], and many other information retrieval systems, such as in "w/2" query at the NEC citeseer system, FBY (followed by) query, etc. The result above is the first nontrivial algorithmic result known for this important problem. (In [19], authors presented algorithms for this problem but in the worst case it involves checking all occurrences of one of the patterns, which is $\Theta(n)$.)

7.3 Large Alphabets. In string matching, there are three alphabet (denoted Σ) models: (i) fixed size, (ii) large size (polynomial in n), and (iii) unbounded. In the main part of this paper, we presented solutions for the document retrieval problems in the fixed size alphabet model. In the unbounded alphabet model (where access to document positions must use comparisons only), matching p against the documents takes $\Omega(m \log m)$ time in the worst case since symbols of p have to be sorted; hence no linear time query is possible. The case that is left to be considered is the large alphabet case. The bottleneck is that there are no known (generalized) suffix tree construction algorithms that use $O(n)$ space and time for construction and support lookup queries on a pattern p (i.e., find u_p in Step 1 of the algorithms we have described) in $O(m)$ time, for the large alphabet case. In particular, the optimal construction in [6] takes time $\Omega(m \log |\Sigma|)$ and the suffix array in [18] takes time $O(m + \log |\Sigma|)$; this is so because they essentially sort the outgoing edges at each node by the first alphabet symbol on the string labeling the edge, and comparisons are used to navigate at each node.

A simple trick will fix this problem. Observe that there are only $O(n)$ tuples given by $(u, \sigma_v[|\sigma_u| + 1]) \rightarrow v$ for each edge (u, v) in the generalized suffix tree. We hash these tuples in $O(n)$ space and use hashed lookup while navigating from any node u. As a result, we get the following which is of independent interest.

THEOREM 7.3. *There exists a randomized $O(n)$ time and space generalized suffix tree construction algorithm with large alphabet set such that finding u_p for given pattern p takes time $O(m)$ with high probability. Using perfect hashing, the preprocessing time becomes $O(n \log n)$ [12], and query time will be deterministic.*

All our reductions for document retrieval problems work optimally for the large alphabet case too, and hence are efficient given the theorem above. ∎