DOCUMENT RETRIEVAL WITH WILDCARDS

by

Clinton Morrison

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Computer Science, Honours Co-op

at

Dalhousie University
Halifax, Nova Scotia
July 2016

Table of Contents

List of Figures

Abstract

Document retrieval is a fundamental information retrieval problem with many applications including web search. In document retrieval problems, the challenge is to index a collection of documents $d_1, d_2, ...d_D$ so that we can efficiently retrieve relevant documents, with respect to a query pattern $P[1..m]$. Every document, as well as P, is a string over some alphabet Σ. Many different document retrieval problems have been studied. In the *top-k retrieval* problem, only the k most relevant documents must be returned. Here, relevance is measured by a score function. One common measure is *term frequency*, the number of occurrences of P in the document. In the *document listing* problem, the goal is to list all of the documents that contain P.

Efficient solutions to these two document retrieval problems are known. However, recent interest in approximate text indexing has motivated new problems. In one kind of approximate text matching, the query pattern P is allowed to contain *wildcard* characters, denoted by 'ϕ'. Each ϕ can match any single character in Σ.

In this thesis we consider the top-k retrieval and document listing problems, where P contains a single wildcard. We present an $O(n \lg n)$-word index for top-k document retrieval. Document relevance is scored by term frequency. The index can answer queries in $O(m + (\frac{k \lg^2 D}{\lg n} + k) \lg \lg n)$ time. Here, n is the total length of the document collection and m is the length of the query pattern. We also give the first index for the document listing problem with optimal $O(m + \text{docc})$ query time, where docc is the number of documents returned. The index occupies $O(n \lg n)$ words of space.

Acknowledgements

I would like to sincerely thank my supervisor, Meng He, for his time, support, and guidance.

Chapter 1

Introduction

Document retrieval is an important and fundamental information retrieval problem. The goal is to index a collection of documents to support efficient retrieval. Usually a query pattern is given and the relevant documents from the collection must be returned. Document retrieval problems have many applications including web search, database systems and bioinformatics.

Many versions of the problem have been studied. In *top-k retrieval*, given a query pattern $P[1..m]$, we want to return only the k most relevant documents. Many methods for scoring relevance are possible. For example, the number of occurrences of the pattern in a document may be used as a measure of its relevance. This is called the *term frequency* of the pattern. Similarly, in the *document listing problem*, the goal is to return the documents that contain at least one occurrence of the pattern P.

One common approach to these problems is an *inverted index* [23]. Here, we have a set of words and for each word we store a list of documents that contain the word. However, this method requires that the text has a logical notion of words to index. This idea does not apply to text that contains DNA data. Furthermore, there are many cases where supporting arbitrary queries would be desirable. This has motivated the development of suffix trees and arrays, which have been successfully applied to many document retrieval problems, e.g. [16].

These classic approaches only support finding the exact occurrences of a pattern in a document. In many cases, we would like to support more approximate matching of a pattern. Many kinds of approximate search have been considered. In one variant, the query pattern may contain a special *wildcard* character, denoted by ϕ. This character can match any character in the alphabet. For example, the pattern "ϕbc" would match "abc", "bbc", "cbc", etc. This kind of approximate matching is especially applicable to bioinformatics, where single nucleotide polymorphisms (SNPs) [7] can be modeled by wildcards. Specifically, there are cases where certain DNA base pairs vary throughout a population. This poses a challenge for aligning DNA sequences. Wildcard pattern matching has been used to support alignment where the sequence contains SNPs [15].

Text indexing with wildcards in the pattern has been studied in depth. However, extending document retrieval indexes to support wildcards has proven to be difficult. Efficient solutions to many document retrieval problems are still not known. In this thesis, new indexes that support a single wildcard in the query pattern are developed.

1.1 Our Results

We present several results for document retrieval problems where the query pattern contains one wildcard, as summarized in the following list. Here, n is the total length of the documents, m is the length of the query pattern, D is the number of documents in the collection, and σ is the size of the alphabet.

- We present the first top-k index which supports patterns containing a single wildcard. Our index achieves a query time of $O(m + (\frac{k \lg^2 D}{\lg n} + k) \lg \lg n)$ and uses $O(n \lg n)$ words of space.

- We also present a new solution to the document listing problem where the pattern may contain 1 wildcard. Our index uses $O(n \lg n)$ words but achieves an optimal query time of $O(m + \text{docc})$, where docc is the number of documents that match P. A recent result from Lewenstein et al. [12] used only $O(n)$ words but had a query time of $O(m + \sigma\sqrt{\lg \lg \lg n} + \text{docc})$.

Chapter 2

Related Work

2.1 Text Indexing with Wildcards

Text indexing is a classic problem that has been studied in depth. The goal is to index text $T[1..n]$ over some alphabet Σ to efficiently support finding occurrences of a query pattern $P[1..m]$ in T.

Many generalizations of this classic problem have been studied. There has been interest in a variety of approximate text indexing problems. In one generalization query patterns are allowed to contain special wildcard characters, denoted by ϕ. The wildcard character may match any single character in Σ. This particular problem is partially motivated by applications to bioinformatics. As discussed in the introduction, one such application is DNA sequence alignment where the sequence contains SNPs.

One of the early results on the problem was by Cole et al. [3]. They gave an $O(n \lg^j n)$-word index with a query time of $O(m + 2^g \lg\lg n + \text{occ})$. Here, occ denotes the number of occurrences of P in the text, j is the maximum number of wildcards allowed in the query and g is the actual number of wildcards in the query.

This result was generalized by Bille et al. [1]. They presented a $O(n \lg n \log_\beta^{j-1} n)$-word solution with query time $O(m + \beta^g \lg\lg n + \text{occ})$, $2 \leq \beta \leq \sigma$. Setting $\beta = 2$ gives Cole's index.

Later Lewenstein et al. showed how the space cost of Cole's index could be improved to $O(n \lg^{j-1} n)$ words while maintaining the same query time [13]. Another index by Lewenstein et al. [14] takes only $O(n)$ words and can answer queries in $O(m + \sigma^g \sqrt{\lg\lg\lg n} + \text{occ})$ time. They also gave a $O(n)$-bit index with $O((m + \sigma^g + \text{occ}) \lg^\epsilon n)$ query time. While the query time is worse, even the space usage of $O(n)$-word indexes has been considered impractical [13] when the text is very long.

Another research direction has focused on the case in which wildcards appear in the text but not in the pattern. Cole's index can support wildcards in both the text and pattern in the same time and space with some modifications [3]. Much of the subsequent work on this problem focused on developing lower space indexes. To review recent results, some additional notation is required. Text that contains $d+1$ text segments separated by d groups of wildcards can be written as $T[1..n] = T_0\phi...\phi T_1\phi...\phi T_d$. We let γ denote the total number of occurrences of text segments $(T_0, T_1, ...T_d)$ in P. Additionally, occ_1 denotes the number of matches of P in T where the matched text contains no wildcards and occ_2 is the number of matches where the text contains a single group of wildcards. k denotes the number of wildcards in the text, and $\hat{d}$ is the number of distinct lengths for groups of consecutive wildcards.

Lam et al. [11] presented a linear space index with a query time of $O(m^2 \lg n + m \lg^2 n + \gamma \lg n + \mathrm{occ})$. Later a succinct space was given by Tam et al. [21]. Their index used $(3 + o(1))n \lg \sigma + O(k \lg n)$ bits and could answer queries in $O(m(\lg \sigma + \min(m, d) \lg k) + \mathrm{occ}_1 \lg^{\epsilon+1} n + \mathrm{occ}_2 \lg^{\epsilon} d + \gamma)$ time. Recent work achieves compressed space. An index by Hon et al. [7] has $O(m(\lg^{1+\epsilon} n + \min(m, \hat{d}) \lg D) + \mathrm{occ}_1 \lg^{1+\epsilon} n + \mathrm{occ}_2 \lg^{\epsilon} d + \gamma \lg \gamma)$ query time with $nH_{o(\log_\sigma n)} + o(n \lg \sigma) + O(d \lg n)$ bits. Here, $H_{o(\log_\sigma n)}$ denotes the $o(\log_\sigma n)$-th order empirical entropy of the text.

2.2 Top-k Document Retrieval

The top-k document retrieval problem has been studied extensively and a variety of score functions have been considered. Some possibilities include a static document assigned score (such as PageRank [20]), term frequency, and the proximity of occurrences of the query pattern (term proximity) [8].

A formal study of this problem was originally initiated by Hon et al. [9]. They gave an $O(n \lg n)$-word index with a query time of $O(m + k + \lg n \lg \lg n)$. It only supports term frequency as a score function.

Later Hon et al. [8] gave a more general framework for top-k problems with various time space trade-offs. They considered mainly the case where the score function is term frequency. However, their framework can be easily applied to other score functions. They presented a linear space framework with $O(m + k)$ query time if the documents are returned in unsorted order. If sorted order is required, then the query time is $O(m + k \lg k)$. Additonal time and space trade-offs are also presented. One index uses $2|CSA| + D \lg \frac{n}{D} + O(D) + o(n)$ bits for a query time of $O(t_s(P) +$

$kt_{sa} \lg k \lg^{\epsilon} n)$. $|CSA|$ denotes the size of a compressed suffix array [6] (CSA) in bits, $t_s(P)$ is the time required to compute the suffix range of the pattern P using the CSA, and t_{sa} is the time to retrieve a value from the CSA. Another one of their indexes occupies $|CSA| + n \lg D + o(n \lg D + n \lg \sigma)$ bits and has a query time of $O(t_s(P) + m + \lg^6 \lg n + k((\lg \sigma \lg \lg n)^{1+\epsilon} + \lg^2 \lg n + \lg k))$.

Hon's index has been extended to support other difficult top-k problems. In [8] Hon et al. present an index for the two pattern (2P) retrieval problem with a query time of $O(m_1 + m_2 + \sqrt{nk \lg D} \lg \lg n)$, where m_1 and m_2 are the lengths of the two patterns. Munro et al. [17] showed how the succinct space index could handle term proximity as a score function. Their index uses $|CSA| + o(n)$ bits and has a query time of $O((m + k)\text{polylog } n)$.

Navarro et al. [19] gave an alternate linear space index which can return documents in sorted order in optimal $O(m + k)$ time. Their index supports multiple score functions including term frequency and proximity.

2.3 Document Listing

The document listing problem was originally studied by Matias et al. [16]. Their index occupied linear space and had a query time of $O(m \lg D + \text{docc})$.

This result was later improved by Muturkishnan [18]. He showed how the problem could be reduced to coloured range reporting and gave a $O(n)$-word index with optimal $O(m + \text{docc})$ query time.

Document listing with a single wildcard in the query pattern is a more difficult problem. Recently Lewenstein et al. [12] proposed an index that utilizes the notion of unique prefixes to achieve $O(n)$ words of space and $O(m + \sigma\sqrt{\lg \lg \lg n} + \text{docc})$ query time.

Document counting is a related problem where the goal is to count how many documents contain a pattern. For regular pattern matching, such queries can be answered in $O(m)$ time by simply annotating each node in the suffix tree with the number of unique documents [13]. Limited work has been done for the case where the query pattern may contain a wildcard. Lewenstein et al. [13] gave an $O(n \lg n)$-space index with optimal $O(m)$ query time.

Chapter 3

Preliminaries

3.1 Range Minimum Queries

Range Minimum Query (RMQ) data structures have found a very wide range of applications. Given an array $A[1..n]$ of integers, an RMQ structure can efficiently find the minimum (or maximum) element in some range $A[i..j]$ where $1 \leq i \leq j \leq n$. We use the following result from Fischer et al. [4]:

Lemma 3.1.1 ([4])**.** *Range minimum (maximum) queries on an array $A[1..n]$ of integers can be answered in $O(1)$ time with an index of $2n + o(n)$ bits.*

3.2 Generalized Suffix Trees

Suffix trees, originally proposed by Weiner [22], are a classic text index which stores all of the suffixes of some text $T[1..n]$ in a compact trie. A suffix tree occupies $O(n)$ words and can support a variety of navigation operations.

In [9] Hon et al. describe a generalized version of suffix trees that indexes a collection of documents. For a set of D documents $d_1, d_2, ..., d_D$, the generalized suffix tree (GST) for the document collection is the suffix tree built over the string $T = d_1\$d_2\$...d_D\$$. Here '\$' is a special character that does not appear in the documents. We consider this character lexicographically smaller than all other characters in the alphabet. By constructing the text this way, each suffix in T is unique, even if two documents are identical. We use the GST as described in [8].

Lemma 3.2.1 ([8])**.** *The generalized suffix tree for a set of D documents of total length n occupies $O(n)$ words and supports the following operations in $O(1)$ time:*

- *Navigation by the starting character of an edge.*
- *Finding left-most and right-most leaves in a node's subtree (the suffix range of a pattern).*

- *Finding the preorder rank of a node.*
- *Finding a node, given its preorder rank.*

3.3 Document Arrays with Rank Queries

Document arrays have proven to be a useful tool in document retrieval problems. Suppose we have some text $T[1..n] = d_1\$d_2\$...d_D\$$, and a suffix array SA built over T. A suffix array SA maintains the suffixes of T in lexicographical order so that $SA[i] = j$ if and only if $T[j..n]$ is the i-th smallest suffix. The entries in SA correspond to the leaves of the GST. We say that a suffix $T[x..n]$ is contained in document y if position x in T corresponds to document d_y.

A document array is an array $E[1..n]$ such that $E[i] = r$ if and only if the i-th leaf in the generalized suffix tree corresponds to a suffix in the r-th document. In other words, $E[i]$ indicates which document contains the suffix $SA[i]$. Each suffix is contained in exactly one document because of how the string T is created, as discussed in the previous section.

We illustrate this with a brief example. Consider the documents $\{c, b, a\}$, with document IDs 1, 2, and 3 respectively. Then $T = c\$b\$a\$$.

The suffixes of T are $c\$b\$a\$, \$b\$a\$, b\$a\$, \$a\$, a\$, \$$. In sorted order, the suffixes are $\$, \$a\$, \$b\$a\$, a\$, b\$a\$, c\$b\$a\$$, so the suffix array is $SA = [6, 4, 2, 5, 3, 1]$. Here, $E = [3, 2, 1, 3, 2, 1]$.

Rank queries on E can support many useful operations. $rank_E(r, i)$ returns the number of occurrences of r in $E[1..i]$. We use ideas from [8] and apply [5] to encoding document arrays, which gives the following result.

Lemma 3.3.1 ([5]). *Let $E[1..n]$ be a document array for a set of D documents. E can be maintained in $n \lg D + o(n \lg D)$ bits such that the operation $rank_E(r, i)$ can be done in $O(\lg \lg n)$ time.*

3.4 Bit Vectors with Rank and Select

A bit vector $B[1..n]$ is a binary string of n bits. We define two basic operations on bit vectors. Firstly, $\text{rank}_B(i)$ returns the number of ones in $B[1..i]$. Additionally,

$\text{select}_B(i)$ returns the position of the i-th one in B. Jacobson [10] showed how bit vectors could be indexed with very little extra space to support these two operations in constant time. His result is summarized in the following lemma.

Lemma 3.4.1 ([10]). *A bit vector $B[1..n]$ can represented using $2n + o(n)$ bits such that rank$_B$ and select$_B$ queries can be answered in $O(1)$ time.*

Chapter 4

Top-k Document Retrieval With One Wildcard

Now we consider the problem of top-k document retrieval. Here we are given a set of D documents $d_1, d_2, d_3, ...d_D$ of total length n, all drawn over an alphabet Σ, where $|\Sigma| = \sigma$. Given a query (P, k), we seek to return the k most relevant documents with respect to some pattern P. Here, $P[1..m] = P_1 \phi P_2$, where ϕ is a wildcard that can match any character in Σ. A variety of score functions for calculating relevance are possible. Term frequency is a very common score function which counts how many occurrences of a pattern occur in a document. We let $\mathrm{TF}(P, d_r)$ denote the number of occurrences of the pattern P in the document d_r. We generalize this to apply to patterns with a single wildcard as follows:

$$\mathrm{SCORE}(P_1 \phi P_2, d_r) = \sum_{\alpha \in \Sigma} \mathrm{TF}(P_1 \alpha P_2, d_r)$$

Top-k indexes may return the top-k document IDs in sorted order (by relevance) or in unsorted order. For all of the indexes described in this thesis, documents are returned in unsorted order. Sorted order can always be returned by sorting the final k results by score. This adds a $O(k \lg k)$ term to the query time.

To create a top-k index that supports wildcards, we combine a classic wildcard index by Cole et al. [3] with ideas from Hon et al. [8].

4.1 Cole's Index

We begin with a review of Cole's wildcard index. His index supports finding all the occurrences of a pattern $P[1..m]$ in some text $T[1..n]$ where the $P[1..m]$ contains a bounded number of wildcards [3]. The non-wildcard characters in P and T are drawn from an alphabet of size $|\Sigma| = \sigma$. We only need to handle query patterns with a single wildcard, so we describe a version of the index for just one wildcard.

If the pattern does not contain any wildcards, occurrences of P can be found efficiently in $O(m + \mathrm{occ})$ time using a regular suffix tree. If P contains a wildcard, then there are σ possible patterns we need to look for in the suffix tree. This leads to a query time of $O(m\sigma + \text{occ})$. To improve the query time, every node in the suffix tree can be augmented with wildcard links. Each wildcard link could point to a new tree which includes all of the possible suffixes that could result from matching a single wildcard in that position. This is all of the suffixes related to that node, except that we skip the first character of each. Hence, when a wildcard is encountered, the wildcard link can be followed. This results in a query time of $O(m + \text{occ})$ but has a prohibitive space cost of $O(n^2)$ words.

Cole's index is based on this idea, but the extra trees only include some of the possible suffixes to reduce the space cost. It is based on the heavy path decomposition of the tree. A child of a node is said to be *heavy* if it contains more leaves in its subtree than the other children. Likewise, the other children are considered *light*. If there is a tie, a single child can be arbitrarily designated as the heavy. Figure 4.1 shows an example of this. Here, u is some node in the tree, the green nodes are light children, and the red node is the heavy child of u. The heavy child has three leaves in it's subtree, while the light children all of 2 or less leaves.

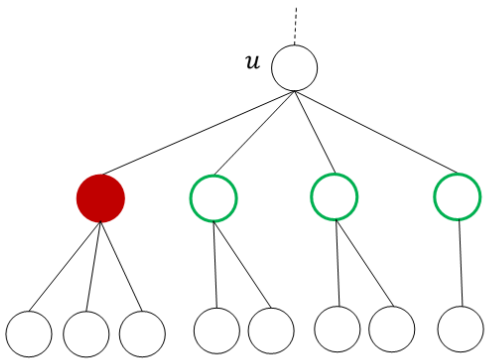

Figure 4.1: The heavy and light children of some node u

In Cole's index, every node in the tree is augmented with a wildcard link. Each wildcard links points to another suffix tree called a *sidetree*. For a node v, sidetree(v) is a suffix tree that contains only certain suffixes related to the light children of v. For each light child u of v, the sidetree contains all of the suffixes corresponding to u, except that we skip the first character on the edge from v to u. Essentially, the first character on each path is matched by the wildcard.

Figure 4.2 shows an example of a sidetree. In this example, the light children of

u contain the suffixes “abbba”, “abbcd”, “bbc”, and “bbd”. The sidetree contains these suffixes, but skips the first character of each. Therefore, the sidetree contains the suffixes “bbba”, “bbcd”, “bc”, and “bd”.

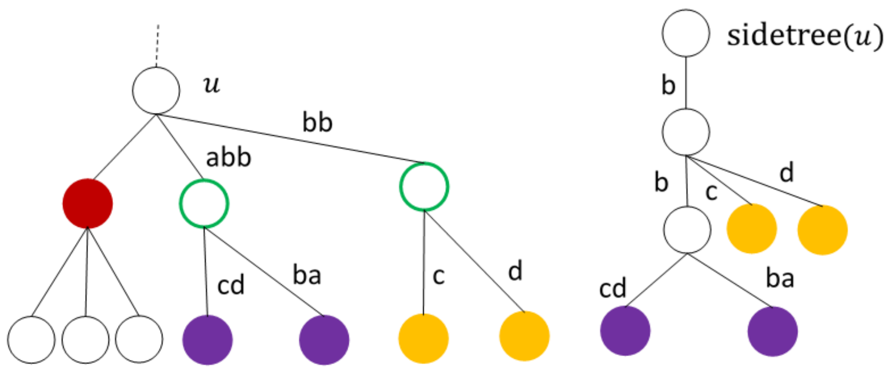

Figure 4.2: The sidetree of some node u

We call nodes in the original tree *level-0* nodes and nodes in a sidetree *level-1* nodes.

To answer queries, matching in the suffix tree is done normally until a wildcard in the pattern is encountered. While following a path in the suffix tree, the next character in P is compared to the next character on the edge's label. If the next character in P is a wildcard, then it matches any character on the edge and matching can continue normally. If a wildcard encountered after reaching some node v (immediately after the last character on an edge is matched), then the search branches. First, follow the wildcard link at v and continue matching the pattern in sidetree(v). Also try to follow the path to the heavy child of v. This procedure always leads to at most two nodes in the tree. We call such nodes *locus nodes*. The leaves of these nodes correspond to all of the occurrences of P in the text.

Since there are at most 2 possible paths to traverse with one wildcard this index can find both locus nodes in $O(m)$ time. We refer to [3] and [13] for the analysis of the space cost of the index. The number of nodes in all sidetrees can be bounded by $O(n \lg n)$. The results are summarized in the following lemma.

Lemma 4.1.1. *With an index of $O(n \lg n)$ words, all locus nodes of a pattern $P[1..m]$ containing at most one wildcard can be located in the augmented tree in $O(m)$ time.*

4.2 Top-k with Cole's Index

Our index is based on the top-k document retrieval framework proposed by Hon et al. [8]. We show that it is possible to combine Cole's wildcard index [8] with this top-k framework to efficiently support queries with wildcards in the pattern.

Let $T[1..n] = d_1\$d_2\$...d_D\$$ be the string obtained by concatenating the documents together. We first build a generalized suffix tree (GST) over T. We then augment this suffix tree with the first level of Cole's index. Every node in the tree is augmented with a wildcard link that points to that node's sidetree. The augmented GST has $O(n \lg n)$ nodes because Cole's index has $O(n \lg n)$ nodes. The overall space required is also $O(n \lg n)$ by Lemma 4.1.1.

We also maintain a document array $E[1..O(n \lg n)]$ over all the leaves of the new tree as in Lemma 3.3.1. The augmented GST has $O(n \lg n)$ leaves, but rank queries on E still take $O(\lg \lg (n \lg n)) = O(\lg \lg n + \lg \lg \lg n) = O(\lg \lg n)$ time.

To handle top-k queries, we build some additional data structures. The idea is to annotate certain nodes with partial precomputed answers. We do this using the same techniques described in Section 5 of [8]. Specifically, each structure we build can help answer top-k queries for a specific value of k. Instead of doing this for all values of k, we only store them for the powers of 2 up to D, e.g. $k = 1, 2, 4, 8, ...D$. We call each structure a tier. Therefore, tier-i stores some precomputed answers for top-2^i queries, for $0 \leq i \leq \lceil \lg D \rceil$. There are $O(\lg D)$ tiers. When looking up an answer, the structure for the nearest higher value of k can be used.

Now, we describe how a tier is built. We consider the i-th tier. The others are constructed in the same way. For tier-i, we must build structures to answer top-2^i queries. Matching a pattern with Cole's index can result in up to two locus nodes (every possible pattern corresponds to a pair of locus nodes). Therefore, to help answer top-2^i queries, we could store precomputed answers for some pairs of nodes. It would require too much space to store precomputed information for all pairs of nodes, so we mark a subset of nodes, and only precompute answers for pairs of marked nodes.

For each tier i, we designate some nodes in the GST as *tier-i marked nodes.* To determine which nodes should be marked, we apply Hon's marking scheme [8]. Here, nodes in the tree are marked using the following rules:

1. Let g be the grouping factor. Partition leaves of the tree into groups of g_i.

2. Mark the lowest common ancestor (LCA) of the first and last node in each group.

3. Mark the LCA of all pairs of marked nodes.

4. Mark the first and last leaves of the subtree rooted at each marked node.

We use g_i to denote the grouping factor used to mark nodes in tier-i. It is important to note that each tier is a separate structure. Nodes being marked in one tier does not effect if they are marked in a different tier. Further, it is not required that each tier have the same grouping factor (since they are separate structures). We will later show how to pick values for each grouping factor to minimize the running time. As already discussed, we will store precomputed top-2^i lists for marked nodes in tier-i. In [8] Hon proves the following properties about the marking scheme:

Lemma 4.2.1 ([8]). *For a tree on $O(n)$ nodes with a grouping factor of g_i, the marking scheme has the following properties:*

- *$O(n/g_i)$ nodes are marked*

- *If y is an unmarked node and x is the highest marked descendant of y, the number of leaves in the subtree of y that are not in the subtree of x is at most $2g_i$.*

In general, Cole's index may result in up to two locus nodes for query patterns with a single wildcard. Therefore, we store precomputed answers for all possible pairs of marked nodes in each tier. To handle the case where there is only one marked node, we define a special imaginary node called the *null node*, denoted by $\emptyset$. This node does not exist in the tree, but acts as a placeholder to store top-k lists for individual nodes. If v is a tier-i marked node, then the tier-i top-2^i list for v is associated with the pair of nodes $(v, \emptyset)$.

For any tier i, there are $O(\frac{n \lg n}{g_i})$ marked nodes and $O(\frac{n^2 \lg^2 n}{g_i^2})$ pairs of marked nodes. However, we do not need to store answers for all pairs because only some pairs are possible.

If there are two locus nodes, one must be in level-0 (which contains only $O(n)$ nodes) and one must be in level-1 (which contains $O(n \lg n)$ nodes). To understand this, consider how the search branches when the wildcard is encountered. On one path, we follow the wildcard link and continue matching in the sidetree. This may lead to one level-1 locus in the sidetree. We also follow the heavy path and continue matching. Since there was only one wildcard in P, no more wildcard links will be followed. Consequently, any locus node found this way must be in level-0. Therefore, we only need to store answers for $O(\frac{n^2 \lg n}{g_i^2})$ pairs.

Storing a top-k list requires $k \lg D$ bits, and $k = 2^i$ for tier-i. We store a different structure for $O(\lg D)$ different tiers. Now we can pick a value for each g_i to guarantee that the overall space cost is still $O(n \lg n)$ words. To achieve this, we need to make sure that all the lists for a single tier occupy only $O(\frac{n \lg^2 n}{\lg D})$ bits. This is because there are $O(\lg D)$ tiers and $\frac{n \lg^2 n}{\lg D} \lg D = n \lg^2 n$ bits $= n \lg n$ words.

Since there are $O(\frac{n^2 \lg n}{g_i^2})$ possible pairs of marked nodes for tier-i, storing the lists for tier-i requires $O(\frac{(n^2 \lg n)(2^i \lg D)}{g_i^2})$ bits. To find an appropriate value for g_i, we set this equal to our space requirement:

$$\frac{n \lg^2 n}{\lg D} = \frac{n^2 2^i \lg n \lg D}{g_i^2}$$

The smallest g_i value that maintains $O(n \lg n)$-word overall space is:

$$g_i = \sqrt{\frac{n 2^i \lg^2 D}{\lg n}} = \sqrt{\frac{n 2^i}{\lg n}} \lg D$$

Since this table occupies $O(n \lg n)$ words, the space of the entire index can be bounded by $O(n \lg n)$ words.

Note that by choosing a different grouping factor for each tier, the grouping factor for tier-i contains a $\sqrt{2^i}$ factor. If we had to use the same grouping factor for every node this would not be possible. Instead, we would have to use the highest value of i to maintain $O(n \lg n)$-word space. Therefore, the grouping factor would have to contain a $\sqrt{D}$ factor instead of a $\sqrt{2^i}$ factor since $2^i = \Theta(D)$ for the largest i. As will be discussed in the following section, $\sqrt{2^i} = \Theta(\sqrt{k})$. This will allow the overall running time to contain a $\sqrt{k}$ factor instead of a $\sqrt{D}$ factor, which is a clear improvement since $k \leq D$.

4.3 Answering Queries

The index can answer top-k queries $(P_1 \phi P_2, k)$ in two steps. First, locus nodes in the augmented GST corresponding to occurrences of the query pattern are found. This can be done as described in Section 4.1. Each regular character in the pattern is matched normally. If the next character to match is a wildcard and a wildcard link is available in the tree, we follow the wildcard link and continue matching normally. We also follow the heavy path of the node and continue matching normally. Finding all locus nodes takes $O(m)$ time by Lemma 4.1.1.

Consequently, up to 2 locus nodes can be found when the pattern contains a single wildcard. There may be one or two locus nodes. If there are two, one must be in the level-0 part of the tree and the other must be in level-1 (because a single wildcard link was followed). We call these nodes u_1 and u_2 respectively. If there was only one locus node u_1, we represent it as the pair of nodes $(u_1, \emptyset)$.

Now we must find the top-k documents corresponding to these two locus nodes. We do not have precomputed top-k lists for all possible k values. Instead, we compute $z = \lceil \lg k \rceil$. Now, 2^z is the smallest power of 2 that is larger than the query k value. Next, we find the highest tier-z marked descendants, u_1^* and u_2^*, of the two locus nodes (or just u_1^* if there was only one locus). This can be done in $O(1)$ time if we build a bit vector that encodes which nodes are marked for each tier, as in [8]. Suppose $B_{g_i}[1..O(n \lg n)]$ has an entry for every node in the augmented GST, such that $B_{g_i}[x] = 1$ if and only if the node with preorder rank x (the x-th node encountered in a preorder traversal of the tree) is a tier-i marked node. Further, augment the bit vector with rank and select structures as in Lemma 3.4.1. Each bit vector occupies $O(n \lg n)$ bits. We need one for each tier. Since there are $O(\lg D)$ tiers, the overall space is still bounded by $O(n \lg n \lg D)\text{bits} = O(n \lg n)$ words.

The preorder rank of highest marked descendant of some node u with preorder rank x can be found in $O(1)$ time with:

$$\text{select}_{B_{g_i}}(\text{rank}_{B_{g_i}}(x) + 1)$$

Given the preorder rank of the marked node, we can get find the node in the GST in $O(1)$ time with Lemma 3.2.1.

We retrieve the precomputed top-2^z list for the pair of marked nodes. Recall that if there was just one locus node, the top-2^z list is stored for the pair $(u_1^*, \emptyset)$. This gives us $O(k)$ candidates because $k = \Theta(2^z)$. These candidates reflect the top-k documents for the subtrees of the two marked nodes, but may not include all of the leaves of the original locus node. We add the document IDs of all the leaves we missed to the candidates set as follows. Let $\text{Leaf}(x \setminus y)$ denote the set of leaves that are contained in the subtree of x, excluding those in the subtree of y. If there are two locus nodes, we check the documents that mark the leaves:

$$\text{Leaf}(u_1 \setminus u_1^*) \cap \text{Leaf}(u_2 \setminus u_2^*)$$

Similarly, if there was only one locus node we just check the leaves $\text{Leaf}(u_1 \setminus u_1^*)$. In either case, there are $O(g_z)$ leaves in each of these ranges (g_z is the grouping factor) by Lemma 4.2.1. There are $O(g_z + k)$ candidate documents.

We do a linear scan of these results and remove any duplicate document IDs. This can be done easily with a bit vector of D bits. We describe a classic procedure for performing this task in the following lemma. Hon's index also uses this idea to remove duplicate document IDs.

Lemma 4.3.1. *Suppose we have a list of $A[1..n]$ of n integers, such that $A[i] < U$ for all $i, 1 \leq i \leq n$. We can create a new list that contains all of the unique elements in A without any duplicates in $O(n)$ time using $O(U)$ extra bits.*

Proof. Initialize a bit vector $B[1..U]$ to all zeros. We iterate over A. For each integer $A[i] = x$, do nothing if $B[x]$ is already set to 1. Otherwise, report the integer and set $B[x]$ to 1. This takes $O(n)$ time since we look at each element in A once. Furthermore, it is easy to see that each unique integer is reported only once.

Finally, we reset all of the bits in B to 0 by iterating through the $O(n)$ integers reported. For each reported integer x, we set $B[x] = 0$. This also takes $O(n)$ time since there are $O(n)$ integers. $\square$

Now that we have a unique list of candidate documents, we must compute the score of each. The term frequency of each document can be computed by counting how many times each occurs in the subtree of each locus node. We note that the sidetrees do contain duplicates of suffixes in the main tree. It may appear that there is a risk of double counting certain suffixes when calculating the term frequency.

However, this is not an issue because the level-1 locus is guaranteed to contain a set of suffixes not in the subtree of the level-0 node (by the properties of Cole's index and the heavy path decomposition.)

Let N_l denote the number of locus nodes. The term frequency of each document d_r can be computed with rank queries as follows:

$$TF(P, d_r) = \sum_{j=1}^{N_l} (\mathrm{rank}_E(ep_j, d_r) - \mathrm{rank}_E(sp_j, d_r))$$

Two rank queries per document are performed. Recall each rank query takes $O(\lg \lg n)$ time by Lemma 3.3.1, so the score for every candidate document can be computed in $O((g_z + k) \lg \lg n)$ time.

Finally, we need to take just the top-k documents from the candidate set. As in Hon's index, we use a linear time selection algorithm (such as [2]) to find the k-th highest scoring document in $O(g_z + k)$ time. Suppose the score is s. Next, we do another linear scan of the list and report all of the documents that have a score greater than s. If we have reported less than k documents, multiple documents might have the score s. We iterate over the collection one more time and report documents with score equal to s until k documents are reported or the end of the candidate list is reached. This also takes $O(g_z + k)$ time.

We set $g_i = \sqrt{\frac{n2^i}{\lg n}} \lg D$. Recall that $k = \Theta(2^z)$. This gives us the following result.

Theorem 4.3.2. *A collection of D documents can be indexed in $O(n \lg n)$ words such that the top-k documents for a query pattern $P[1..m] = P_0 \phi P_1$ can be returned in $O(m + (\sqrt{\frac{nk}{\lg n}} \lg D + k) \lg \lg n)$ time.*

In the following section, we give another index with a better query time and the same space by only marking single nodes instead of pairs of nodes. We remark that while the index we just described is worse in query time, we believe it is more likely to lead to a solution that supports patterns with multiple wildcards in the pattern. A pattern containing h wildcards may result in up to 2^h locus nodes with Cole's index. Storing precomputed top-k lists for some groups of marked nodes might lead to an efficient index. On the other hand, if only individual nodes are marked, then the query time will contain a $O(2^h k)$ term.

4.4 A Faster Index

In the previous section we stored precomputed top-k results for all pairs of marked nodes in each tier. As we discussed, storing results for groups of marked nodes may be helpful in creating a more general index. However, for the case where there is only one wildcard in the pattern, better results can be obtained if we only store precomputed top-k lists for individual marked nodes.

As in the previous section, we build a GST over the document collection, augment every node with Cole's wildcard links, and build structures to help answer queries for $k = 1, 2, 4, 8, ..., D$. We still apply Hon's marking scheme for to each tier to determine which nodes are marked. However, we only store precomputed top-k results for individual marked nodes (we no longer store precomputed results for pairs of marked nodes). This greatly reduces the space required and allows us to choose a smaller grouping factor for each tier.

For each i, tier-i contains $O(\frac{n \lg n}{g_i})$ marked nodes. There are $O(\lg D)$ tiers, and each list requires $O(k \lg D)$ bits. To get an overall space of $O(n \lg^2 n)$ bits:

$$\frac{n \lg^2 n}{\lg D} = \frac{n \lg n}{g_i}(2^i \lg D)$$

Solving the equation gives $g_i = \frac{2^i \lg^2 D}{\lg n}$. This gives us an $O(n \lg n)$ word index. Queries can be answered using a similar strategy as in the previous section. We apply Cole's index to find up to two locus nodes, u_1 and u_2, in $O(m)$ time.

Next, we retrieve the top-2^z lists for the highest tier-z marked descendants of each locus node, where $z = \lceil \lg k \rceil$ and $2^z = \Theta(k)$. The marked descendants can be found in $O(1)$ time as described in Section 4.3.

This gives us a candidate set of size $O(2k) = O(k)$. As before, we add the additional relevant document IDs to the candidate set. These document IDs correspond to leaves outside of the subtrees of the marked nodes:

$$\text{Leaf}(u_1 \setminus u_1^*) \cap \text{Leaf}(u_2 \setminus u_2^*)$$

If there is only one locus node, we only have to consider the leaves in $\text{Leaf}(u_1 \setminus u_1^*)$. By Lemma 4.2.1, there are $O(g_z)$ leaves in the combined set. The candidate set contains $O(g_z + k)$ documents. The rest of the process is identical to the previous index. We scan through each list and remove duplicate document IDs in $O(g_z + k)$ time using Lemma 4.3.1. We compute the score of each document and return the k documents with the highest scores. Since $g_z = \frac{2^z \lg^2 D}{\lg n}$ and $2^z = \Theta(k)$, we get a query time of $O(m + (\frac{k \lg^2 D}{\lg n} + k) \lg\lg n)$. This is summarized in the following theorem.

Theorem 4.4.1. *A collection of D documents of total length n can be indexed in $O(n \lg n)$ words such that the top-k documents for a query pattern $P[1..m] = P_0 \phi P_1$ can be returned in $O(m + (\frac{k \lg^2 D}{\lg n} + k) \lg\lg n)$ time.*

This result is comparable to the running times for other top-k and text indexing problems. While the optimal running time regular top-k retrieval would be $O(m+k)$, top-k with wildcards intuitively appears to be a harder problem. Our index is faster than than the best known indexes for other similar problems. In two pattern (2P) retrieval there are two query patterns and the goal is to find all of the documents that contain both patterns. This is similar to wildcard retrieval in that more than one pattern must be considered. There are essentially σ different patterns to search for in the wildcard problem (the wildcard may be replaced by any character in the alphabet). Hon's recent [8] index for the two pattern top-k problem used $O(n)$-words and could only answer queries in $O(m_1 + m_2 + \sqrt{nk \lg D} \lg\lg n)$ time, where m_1 and m_2 are the lengths of the two patterns.

Chapter 5

Document Listing with One Wildcard

Finally, we consider the document listing problem where the query pattern may contain a single wildcard, $P = P_0\phi P_1$. Given a query pattern P we must list all documents that match P in at least one position. Our approach combines Cole's index [3] with ideas from Muthukrishnan [18] to obtain an index with optimal query time.

5.1 Augmented GST

First, we build a GST over the concatenated text of the set of documents as in Lemma 3.2.1. Next, the GST is augmented with the first level of Cole's index as described in Chapter 4. The resulting tree has $O(n \lg n)$ leaves.

A document array $E[1..O(n \lg n)]$ is constructed so that $E[i] = r$ if and only if the i-th leaf corresponds to a suffix in document r. There are $O(n \lg n)$ entries in E and each takes $O(\lg D)$ bits. The space can be bounded by $O(n \lg n)$ words.

As discussed in Chapter 4, pattern matching with Cole's index will lead to at most two locus nodes, u_1 and u_2. The document IDs of leaves in either subtree correspond to documents that match the pattern. Suppose u_1 has a suffix range of $[s_1, e_1]$ and u_2 has a suffix range of $[s_2, e_2]$. Listing all the documents that match P is equivalent to listing the unique document IDs in $E[s_1..e_1] \cap E[s_2..e_2]$. We apply ideas from Muthukrishnan [18] to do this efficiently.

5.2 Coloured Range Listing

In [18] Muthukrishnan showed how document listing can be reduced to one dimensional coloured range listing. We apply his solution to the problem. An array $C[1..O(n \lg n)]$ is built. C is called the chain array of E. Here, $C[i] = j$ if and only if j is the largest integer such that $E[j] = E[i]$ and $j < i$. If there is no such j then $C[i] = -1$. More intuitively, if $E[i]$ is the first occurrence of a document in E then $C[i] = -1$. Otherwise, $C[i]$ gives the index of the last occurrence of the document that precedes $E[i]$.

Muthukrishnan showed how the chain array could be used to list only the document IDs in a given range. The idea is that we only want to report the first occurrence of each document ID in the range. If the query range is $[L, R]$, then these are precisely the document IDs $E[i]$ where $C[i] < L$.

An RMQ can be used to find all of the elements in $C[L..R]$ that are less than L. To do this, find the minimum element, $C[i]$, in $C[L..R]$. If it is larger than L, stop. Otherwise report $C[i]$ since $C[i] < L$. Next, find the minimum elements in $C[L..(i-1)]$ and $C[(i+1)..R]$. Again, report each minimum if it is less than L. Continue in this way until all elements less than L are reported.

Lemma 3.1.1 gives an $O(n)$-bit index which can answer range minimum queries in $O(1)$ time. The document array E takes $O(n \lg D)$ bits and C requires $O(n \lg n)$ bits. Using these ideas Muthukrishnan obtained the following result.

Lemma 5.2.1 ([18])**.** *Given an array $A[1..n]$ and a query $[L, R]$, all of the k unique integers in $A[L..R]$ can be reported in $O(k)$ time using an $O(n)$-word index.*

It is straight forward to extend this result to work with two ranges over the document array for the augmented GST.

Lemma 5.2.2. *Given an array $E[1..O(n \lg n)]$ and a query $([s_1, e_1], [s_2, e_2])$, all of the k unique integers in $E[s_1..e_2] \cap E[s_2..e_2]$ can be reported in $O(k)$ time using an $O(n \lg n)$-word index.*

Proof. We have an array $E[1..O(n \lg n)]$ and two query ranges $[s_1, e_1], [s_2, e_2]$. We can build the index of Lemma 5.2.1 over E. This takes $O(n \lg n)$ words because there are $O(n \lg n)$ elements in E.

Suppose $E[s_1..e_1]$ contains k_1 unique integers and $E[s_2..e_2]$ contains k_2 unique integers. Let k be the total number unique integers in both ranges combined (integers that appear in the first range, the second range, or both).

Apply Lemma 5.2.1 report all of the k_1 unique documents in $E[s_1..e_1]$. This takes $O(k_1)$ time. Use the Lemma again to report the k_2 unique documents in $E[s_2..e_2]$. This takes $O(k_2)$ time. In this second list, there may be some duplicates that were already reported by the first query.

We combine the two lists to get one list of $O(k_1+k_2)$ elements. Duplicate document IDs can now be removed in $O(k_1 + k_2)$ time using Lemma 4.3.1.

Clearly $k_1 + k_2 \leq 2k$. In the worst case both ranges contain that same set of unique integers, so every document ID is considered twice. Therefore, the overall time required is $O(k_1 + k_2) = O(2k) = O(k)$. □

5.3 Answering Queries

Given a query pattern $P[1..m] = P_0\phi P_1$ we apply Cole's index to find up to two locus nodes in the augmented GST. This takes $O(m)$ time by Lemma 4.1.1. We refer the reader to Chapter 4 for discussion of how pattern matching is done with Cole's index.

The suffix range of both locus nodes ($[s_1, e_1]$ and $[s_2, e_2]$ respectively) are next found in $O(1)$ time using the GST, as described in Lemma 3.2.1. Finally, the unique document IDs in $E[s_1..e_1] \cap E[s_2..e_2]$ are reported in $O(\text{docc})$ time using Lemma 5.2.2. docc denotes the number of documents that match the number of pattern. The same approach can be used if there is just one locus node (there is just one suffix range in E instead of two). This gives the following theorem.

Theorem 5.3.1. *A collection of D documents over an alphabet of size σ can be indexed in $O(n \lg n)$ words so that for all documents containing at least one occurrence of a pattern $P[1..m] = P_0\phi P_1$ can be reported in $O(m+\mathit{docc})$ time. Here, docc denotes the number of documents that contain at least one occurrence of P.*

Chapter 6

Conclusion

In this thesis we presented two new indexes for document retrieval problems where the query pattern contains one wildcard. Our solutions combine Cole's classic wildcard text indexing structures with traditional approaches for each of these problems.

For the top-k problem, we applied techniques from Hon et al. [8]. The general idea is to find all of the locus nodes of the pattern using Cole's index. Next, the top-k documents can be found by generating a candidate set of documents. The candidate set size is kept small by maintaining precomputed top-k answers for some nodes in the tree. Next, the score of each candidate is computed. Finally, the top-k highest scoring documents are returned.

For the document listing problem we combined Cole's index with ideas from Muthukrishnan [18] to obtain an index with optimal query time. As with the other solution, the locus nodes for the query pattern are found using Cole's index. A coloured range reporting structure is built over the document array. This structure is used to list all of the unique document IDs in the suffix range of each locus node. Finally, duplicate document IDs are removed.

Document retrieval with wildcards has proven to be a challenging problem. There are still many open problems, and many ways to extend this work. We discuss some ideas in the following list.

- Our top-k index only supports one wildcard in the pattern. It would be natural to attempt to extend them to work with multiple wildcards, especially since the underlying wildcard text indexes support multiple wildcards. The challenge is that the number of possible locus nodes is exponential in the number of wildcards. The query time is related to how many precomputed sets of top-k answers we can store. Many possible combinations of locus nodes could result from matching a pattern with multiple wildcards. Storing precomputed answers for all possible combinations of marked nodes is not feasible. Nevertheless, our index in Section 4.4 could be a good starting point for solving this problem.

- In our discussion of the top-k problem, we only considered using term frequency as a score function. These ideas could be applied to support other score functions.

- Coloured range listing structures support listing all of the distinct colours in a single range. It would be interesting to see if these structures could be extended to list all of the distinct colours in multiple ranges efficiently. This would allow our document listing index to handle multiple wildcards in the pattern.

- Work continues to be done on indexing text to support wildcard matching. New improvements and trade-offs for this problem could support better document retrieval indexes. Text indexes with lower space requirements could also be considered.

- Lower bounds on most wildcard document retrieval problems have not been proven. It would be interesting to find lower bounds for these problems and compare existing results.

- Wildcard matching is related to other approximate matching problems including Hamming distance and edit distance. The techniques used in this thesis might apply to these related problems.

Bibliography

[1] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory of Computing Systems*, 55(1):41–60, 2014.

[2] Manuel Blum, Robert W Floyd, Vaughan Pratt, Ronald L Rivest, and Robert E Tarjan. Time bounds for selection. *Journal of computer and system sciences*, 7(4):448–461, 1973.

[3] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 557:91–100, 2004.

[4] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[5] Alexander Golynski, J Ian Munro, and S Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 368–373. Society for Industrial and Applied Mathematics, 2006.

[6] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[7] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V Thankachan, and Jeffrey Scott Vitter. Compressed text indexing with wildcards. *Journal of Discrete Algorithms*, 19:23–29, 2013.

[8] Wing-Kai Hon, Rahul Shah, Sharma V Thankachan, and Jeffrey Scott Vitter. Space-efficient frameworks for top-k string retrieval. *Journal of the ACM (JACM)*, 61(2):9, 2014.

[9] Wing-Kai Hon, Rahul Shah, and Shih-Bin Wu. Efficient index for retrieving top-k most frequent documents. In *International Symposium on String Processing and Information Retrieval*, pages 182–193. Springer, 2009.

[10] Guy Jacobson. Space-efficient static trees and graphs. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pages 549–554. IEEE, 1989.

[11] Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Space efficient indexes for string matching with dont cares. In *International Symposium on Algorithms and Computation*, pages 846–857. Springer, 2007.

[12] Moshe Lewenstein, J Ian Munro, Yakov Nekrich, and Sharma V Thankachan. Document retrieval with one wildcard. *Theoretical Computer Science*, 635:94–101, 2016.

[13] Moshe Lewenstein, J Ian Munro, Venkatesh Raman, and Sharma V Thankachan. Less space: Indexing for queries with wildcards. *Theoretical Computer Science*, 557:120–127, 2014.

[14] Moshe Lewenstein, Yakov Nekrich, and Jeffrey Scott Vitter. Space-efficient string indexing for wildcard pattern matching. *arXiv preprint arXiv:1401.0625*, 2014.

[15] Ruibang Luo, Chang Yu, Chi-Man Liu, Tak-Wah Lam, Thomas Wong, Siu-Ming Yiu, Ruiqiang Li, and Hing-Fung Ting. Efficient snp-sensitive alignment and database-assisted snp calling for low coverage samples. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, pages 163–169. ACM, 2012.

[16] Yossi Matias, S Muthukrishnan, Süleyman Cenk Sahinalp, and Jacob Ziv. Augmenting suffix trees, with applications. In *European Symposium on Algorithms*, pages 67–78. Springer, 1998.

[17] J Ian Munro, Gonzalo Navarro, Jesper Sindahl Nielsen, Rahul Shah, and Sharma V Thankachan. Top-k term-proximity in succinct space. In *International Symposium on Algorithms and Computation*, pages 169–180. Springer, 2014.

[18] S Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666. Society for Industrial and Applied Mathematics, 2002.

[19] Gonzalo Navarro and Yakov Nekrich. Top-k document retrieval in optimal time and linear space. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1066–1077. SIAM, 2012.

[20] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[21] Alan Tam, Edward Wu, Tak-Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *International Symposium on String Processing and Information Retrieval*, pages 39–50. Springer, 2009.

[22] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[23] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.