String Retrieval for Multi-pattern Queries*

Wing-Kai Hon[1], Rahul Shah[2],
Sharma V. Thankachan[2], and Jeffrey Scott Vitter[3]

[1] Department of CS, National Tsing Hua University, Taiwan
wkhon@cs.nthu.edu.tw
[2] Department of CS, Louisiana State University, USA
{rahul,thanks}@csc.lsu.edu
[3] Department of EECS, The University of Kansas, USA
jsv@ku.edu

Abstract. Given a collection $\mathcal{D}$ of string documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n, which may be preprocessed, a fundamental task is to retrieve the most relevant documents for a given query. The query consists of a set of m patterns $\{P_1, P_2, ..., P_m\}$. To measure the relevance of a document with respect to the query patterns, we may define a score, such as the number of occurrences of these patterns in the document, or the proximity of the given patterns within the document. To control the size of the output, we may also specify a threshold (or a parameter K), so that our task is to report all the documents which match the query with score more than threshold (or respectively, the K documents with the highest scores).

When the documents are strings (without word boundaries), the traditional inverted-index-based solutions may not be applicable. The single pattern retrieval case has been well-solved by [14,9]. When it comes to two or more patterns, the only non-trivial solution for proximity search and common document listing was given by [14], which took $\tilde{O}(n^{3/2})$ space. In this paper, we give the first linear space (and partly succinct) data structures, which can answer multi-pattern queries in $O(\sum |P_i|) + \tilde{O}(t^{1/m} n^{1-1/m})$ time, where t is the number of output occurrences. In the particular case of two patterns, we achieve the bound of $O(|P_1| + |P_2| + \sqrt{nt} \log^2 n)$. We also show space-time trade-offs for our data structures. Our approach is based on a novel data structure called the *weight-balanced wavelet tree*, which may be of independent interest.

1 Introduction

Given a collection $\mathcal{D}$ of string documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n, the *document retrieval problem* is to pre-process this collection, so that when a set of patterns comes as a query, we can efficiently output those documents relevant to this query in some ranked order. This is one of the most fundamental problems in Information Retrieval, and finds applications in web search engines, SQL databases,

* This work is supported in part by Taiwan NSC Grant 96-2221-E-007-082 and US NSF Grants CCF-1017623 and CCF-0621457.

E. Chavez and S. Lonardi (Eds.): SPIRE 2010, LNCS 6393, pp. 55–66, 2010.

and genome alignment tools. The ranking of documents is done using relevance scores, which might depend on the frequencies of the patterns in the document, the proximity of occurrences of the patterns as they appear in the document, or simply the static (query-independent) PageRank [2] of the document.

Traditionally, the input documents are split into words and then an inverted index is built over such data. However, in the case of genome data or some Asian texts, there may be no natural word demarcation, so that the inverted index may require too much space, or may only provide limited searching capabilities. Alternatively, full-text indexes like suffix trees and suffix arrays have been successfully used when the query consists of only one pattern [12,14,17,18,9].

For two-pattern queries (listing all documents with both the patterns), the only known solution was given by [14], which requires $\tilde{O}(n^{3/2})$ space and answers a query in $O(|P_1| + |P_2| + \sqrt{n} + t)$ time, where P_1 and P_2 are the query patterns, and t is the size of the output[1]. Recently, Cohen and Porat [3] proposed an elegant framework for the set-intersection problem, through which the index space is reduced greatly to $O(n \log n)$, while admitting slightly worse query time bounds which, instead of having $\sqrt{n} + t$ term, aims to achieve a $\sqrt{nt}$ term. They achieve $O(|P_1| + |P_2| + \sqrt{nt} \log^{2.5} n)$ query bounds for common document listing problem. In this paper, we build a framework which can handle string retrieval under various relevance notions. Our solution is based on the wavelet-tree-based document retrieval scheme given by Välimaki and Mäkinen [18] augmented with other primitives. We introduce a new version of wavelet tree called *weight-balanced wavelet tree* and deploy a multi-way search paradigm on it [3]. Consequently, we show that the index space can further be reduced to linear, while the query can be answered *more quickly* in $O(|P_1| + |P_2| + \sqrt{nt} \log^{1.5} n)$ time. In addition, our framework can easily be extended to handle queries with more than two patterns.

To avoid retrieving many unwanted results, we also focus on retrieving the most *relevant* documents, where either the K highest-scoring documents, or those with scores above some threshold, are reported. This is in contrast to most of the earlier work, which focussed on retrieving all the possible documents.

One of the pressing issues for suffix-tree-based solutions is their space usage. For instance, a simple implementation of Muthukrishnan's index [14] for frequency based retrieval for single pattern takes about 250 times the original text [20] in practice. Such indexes quickly become impractical for massive data sets. With the advent of the field of succinct data structures, compressed text indexes have been developed which can now compete in terms of space with the inverted indexes (which are typically only 5% overhead over the actual text size). In this paper, we also show how to achieve space-time tradeoffs so that our indexes can be made partly succinct.

1.1 Comparison with Previous Work

Document Retrieval Problems on Single Pattern (P):

[1] The notation $\tilde{O}$ ignores poly-logarithmic factors. Precisely, $\tilde{O}(f(n)) \equiv O(f(n) \log^{O(1)} n)$.

- **Document Listing:** List all the documents which contain P.
- **K-mine:** With an extra online parameter K, return all documents which contain at least K occurrences of P.
- **K-repeats:** With an extra online parameter K, list all documents which contain a pair of occurrences of P that are at most distance K apart.

Muthukrishnan [14] gave a linear space structure for the document listing problem which can answer queries in optimal $O(|P| + t)$ time, which improves the previous $O(|P| \log |\mathcal{D}| + t)$ time index by Matias et. al. [12], where t denotes the size of the output. He also gave indexes that answer K-mine and K-repeats in optimal $O(|P| + t)$ time. The space requirements for both cases is $O(n \log n)$.

Sadakane [17] showed how to solve document listing problems using succinct data structures. Similar work done by Välimaki and Mäkinen [18] shows how to achieve partly succinct space by maintaining a wavelet tree of a document array. Here, document array is an array $D_A[1..n]$ such that $D_A[i]$ stores the id of the document where the ith smallest suffix belongs to. For K-mine and K-repeats problems, Hon et. al. [9] gave a unified framework which solves these problems in optimal $O(|P| + t)$ time, while using only linear space. For the top-K version of the above problems, they gave a linear space index with $O(|P| + K \log K)$ query time. Further they showed how to solve these problems in succinct space.

Document Retrieval Problems on Two Patterns (P_1 and P_2):

- **Document Listing:** We need to list all documents where both P_1 and P_2 occur at least once. Muthukrishnan [14] gave an $\tilde{O}(n^{3/2})$-space index that answers a query in $O(|P_1| + |P_2| + \sqrt{n} + t)$ time. Recently, Cohen and Porat [3] proposed an index taking $O(n \log n)$ words of space, based on their solution to the fast set intersection (FSI) problem. They showed that the document listing problem can be reduced to indexing a $\log n$-partition of the suffix ranges and then solving $\log^2 n$ online set intersection queries. Their solution takes $O(n \log n)$ space and $O(|P_1| + |P_2| + \sqrt{nt} \log^{2.5} n)$ query time, where as our new index takes $O(n)$ space and $O(|P_1| + |P_2| + t \log n + \sqrt{nt} \log^{1.5} n)$ time.
- **K-mine:** We need to list all documents that contain at least K total occurrences of P_1 and P_2. Unfortunately, the above suffix-range partition technique for the document listing problem [3] does not work. To the best of our knowledge, no non-trivial solution (except by computing the number of occurrences of P_1 and P_2 explicitly for each document that contains both P_1 and P_2) is known. In the paper we propose an linear space index with $O(|P_1| + |P_2| + t \log n + \sqrt{nt} \log^2 n)$ query time for this problem.
- **K-repeats:** We need to list all documents that contain a pair of occurrences of P_1 and P_2 that are at most K distance apart. Precisely, let $mindist_i(P_1, P_2)$ denote the minimum distance between an occurrence of P_1 and an occurrence of P_2 in document d_i. Our target is to list all documents d_i with $mindist_i(P_1, P_2) \leq K$. Muthukrishnan [14] showed how to reduce this problem to the *close common colors problem*. Consequently, he gave an index with $O(n^{3/2} \log n)$-space, which can answer the query in $O(|P_1| + |P_2| +$

$\sqrt{n}\log n + t)$ time. In the paper we propose an linear space index with $O(|P_1| + |P_2| + K\log^2 n + \sqrt{nK}\log^2 n)$ query time for this problem.

2 Preliminaries

We introduce some data structures which form the building blocks of our indexes. Throughout the paper, we use $\mathcal{D}$ to denote a given collection of documents $\{d_1, d_2, ..., d_{|\mathcal{D}|}\}$ of total length n. The patterns which come as online query are denoted by P_1 and P_2. We assume that the characters in documents as well as the patterns are taken from an alphabet set Σ of size σ.

2.1 Suffix Trees and Suffix Arrays

Given a text $T[1...n]$, a substring $T[i...n]$ with $1 \leq i \leq n$ is called a suffix of T. The lexicographic arrangement of all n suffixes of T in a compact trie is called the *suffix tree* of T [19], where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in the suffix tree is labeled by a character string and for any node u, path(u) is the string formed by concatenating the edge labels from root to u. For any leaf v, path(v) is exactly the suffix corresponding to v. For a given pattern P, a node u is defined as the *locus node* of P if it is the closest node to the root such that P is a prefix of path(u); such a node can be determined in $O(|P|)$ time. Similarly *generalized suffix tree* (GST) is a compact trie which stores all suffixes of all strings in a given collection $\mathcal{D}$ of strings. For the purpose of our index, we define an extra array D_A called *document array*, such that $D_A[i] = j$ if and only if the $i$the lexicographically smallest suffix is from document d_j.

Suffix array $SA[1...n]$ of a text T is an array such that $SA[i]$ stores the starting position of the ith lexicographically smallest suffix of T [11]. In SA the starting positions of all suffixes with a common prefix are always stored in contiguous range. The suffix range of a pattern P is defined as the maximal range $[\ell, r]$ such that for all $j \in [\ell, r]$, P is a common prefix of the suffix which starts at $SA[j]$.

2.2 Wavelet Tree

Let $A[1...n]$ be an array of length n, where each element $A[i]$ is a symbol drawn from a set Σ of size σ. The *wavelet tree* (WT) [6] for A is an ordered balanced binary tree on Σ, where each leaf is labeled with a symbol in Σ, and the leaves are sorted alphabetically from left to right. Each internal node W_k represents an alphabet set Σ_k, and is associated with a bit-vector B_k. In particular, the alphabet set of the root is Σ, and the alphabet set of a leaf is the singleton set containing its corresponding symbol. Each node partitions its alphabet set among the two children (almost) equally, such that all symbols represented by the left child are lexicographically (or numerically) smaller than those represented by the right child. For the node W_k, let A_k be a subsequence of A by retaining only those symbols that are in Σ_k. Then B_k is a bit-vector of length $|A_k|$, such

that $B_k[i] = 0$ if and only if $A_k[i]$ is a symbol represented by the left child of W_k. Indeed, the subtree from W_k itself forms a wavelet tree of A_k. Note that B_k is augmented with Raman et al.'s scheme [15] to support constant-time bit-rank and bit-select operations and we do not store A_k's explicitly. The total space requirement of wavelet tree is $n \log \sigma(1 + o(1))$ bits and it can support orthogonal range reporting in $O((|output| + 1) \log \sigma)$ time [10,21].

2.3 Weight-Balanced Wavelet Tree

We propose a modified version of the wavelet tree called *weight-balanced wavelet tree* (WBT), where we maintain the number of 0's and 1's in any bit-vector B_k almost equal. As a result, the two children of a node may not represent equal (or almost equal) distinct number of symbols, but they should represent almost equal total number of symbols. For this, we first consider the symbols in Σ by their number of occurrences (in ascending order) in A, instead of lexicographic order. Consider a node W_k which represents the alphabet $\{\sigma_1, \sigma_2, ...\}$, such that $f_1 \leq f_2 \leq \cdots$, where f_i is the number of occurrences of σ_i in A. Then $\sum f_i = |B_k| = n_k$. The partition of the alphabet for the child nodes is performed as follows. Initialize $n_\ell = n_r = 0$. Pick up the symbol σ with the most occurrences f. If $n_\ell \leq n_r$, we put σ to the left child, and increment n_ℓ by f. Otherwise, if $n_\ell > n_r$, we put σ to the right child, and increment n_r by f. The process continues until all the symbols are distributed. Once B_k of a node is computed, we perform the procedure recursively in the child nodes until the node represents a single symbol. This way of partitioning ensures the following property(proof omitted).

Lemma 1. *Let W_k be a node in WBT at depth δ_k, and B_k denote its associated bit-vector. Let $n_k = |B_k|$. Then we have $n_k \leq 4n/2^{\delta_k}$.* □

An interesting property is that a WBT is inherently compressible(with out compressing the individual bit vectors).

Lemma 2. *The space of a weight-balanced wavelet tree of an array A of size n is $n(H_0(A) + 2)$ bits, where $H_0(A)$ is the 0th-order empirical entropy of A.*

Proof: Let the depth of a leaf corresponding to the symbol σ_i be δ_i. Then σ_i contributes f_i bits in each bit-vector corresponding to the nodes from root to this leaf (excluding the leaf). Hence the contribution of σ_i towards the total space is $f_i \cdot \delta_i$. By Lemma 1, $\delta_i \leq \log(4n/f_i)$. Therefore, the total size of a weight-balanced wavelet tree is at most $\sum f_i \cdot (\log(n/f_i) + 2) = n(H_0(A) + 2)$ bits. □

3 Any-One Index

In this section we describe an index which we call an *Any-One index*. This is a building block for the indexes described in later sections. The input query consists of two patterns P_1 and P_2, and a parameter K. Any-One index for K-mine problem answers whether or not there exists at least one document d_i with

score $= f_{i1} + f_{i2} \geq K$, where f_{ij} denotes the number of occurrences of P_j in d_i. An Any-One index for K-repeats problem answers whether there exists at least one document d_i with $mindist_i(P_1, P_2) \leq K$.

3.1 Index Construction

Let GST be the generalized suffix tree for the collection $\mathcal{D}$, and GSA be the corresponding generalized suffix array. We maintain a document array D_A where $D_A[i] = j$ if the ith lexicographically smallest suffix belongs to d_j. Let WT be the wavelet tree of D_A. We define a parameter $g = \sqrt{n/\beta}$ which is called *group size*. For K-repeats, we also maintain the wavelet trees (WT_i's) over suffix arrays (SA_i's) of individual documents d_i's.

First, starting from left in GST, we group every g contiguous leaves together to form a group. Thus the first group consists of $\ell_1, ..., \ell_g$, the next group consists of $\ell_{g+1}, ..., \ell_{2g}$, and so on, where ℓ_j denotes the jth leaf. The total number of groups is $O(n/g)$. Now for each group we mark the least common ancestor (LCA) of the first and the last leaves. Furthermore, we continue the marking as follows: if two nodes are marked, we mark their LCA also. It can be easily shown that the total number of marked nodes is $O(n/g)$. Now suppose for a marked node u, its subtree contains the leaves $\ell_x, \ell_{x+1}, \ldots, \ell_y$, then we refer the range $[x, y]$ as the *suffix range* corresponding to u.

Lemma 3. *The suffix range $[L, R]$ of any pattern P can be split into a suffix range $[L', R']$ corresponding to some marked node u^*, and two other small ranges $[L, L' - 1]$ and $[R' + 1, R]$ with $L' - L < g$ and $R - R' < g$.* □

Essentially, the suffix range $[L, R]$ of P corresponds to $R - L + 1$ leaves in the GST. This set of leaves can be partitioned into three groups: one that is under the subtree of u^* which contains $R' - L' + 1$ leaves, and the remaining two with those on the left of $\ell_{L'}$ and those on the right of $\ell_{R'}$. We shall refer to the latter two groups of leaves as *fringe leaves*, each group contains fewer than g leaves.

Score Matrix: We make use of score matrices to facilitate the K-mine and K-repeats queries. The score matrix M for each query is a two-dimensional $O(n/g) \times O(n/g)$ matrix with $O(n^2/g^2)$-word space. For K-mine, the matrix M stores the highest possible score value for any document (along with document id) for each pair of marked nodes. Precisely, we set $M(u^*, v^*) = \max_i\{f_{iu^*} + f_{iv^*}\}$, where f_{iu^*} and f_{iv^*} are the number of leaves in the subtree of u^* and v^* whose suffixes are from document d_i. For K-repeats, we store score as the minimum value of *mindist* between P_{u^*} and P_{v^*}, where $P_{u^*} = \mathrm{path}(u^*)$ and $P_{v^*} = \mathrm{path}(v^*)$. Precisely, we set $M(u^*, v^*) = \min_i\{mindist_i(P_{u^*}, P_{v^*})\}$.

Total Space: This index consists of the GST for $\mathcal{D}$ ($O(n)$ words), the wavelet tree of D_A ($O(n)$ words), individual wavelet trees (for K-repeats) WT_i's ($O(\sum |d_i|) = O(n)$ words), and the score matrix ($O(n^2/g^2)$ words). If we choose $g = \sqrt{n/\beta}$, the score matrix takes $O(n\beta)$ words, and hence the total space is $O(n\beta)$.

3.2 Query Answering

The query input consists of two patterns P_1 and P_2, and a parameter K. We search for these patterns in GST, find their locus nodes u_1 and u_2, and obtain their suffix ranges $[L_1, R_1]$ and $[L_2, R_2]$, respectively. From these suffix ranges, we find the suffix ranges $[L'_1, R'_1]$ and $[L'_2, R'_2]$ (as described in Lemma 3), and the corresponding marked LCA nodes u^* and v^*. Now we check for the score matrix value $M(u^*, v^*)$. For K-mine, if $M(u^*, v^*) \geq K$, it will immediately imply that there exists some document d_i with $f_{i1} + f_{i2} \geq K$. However, if $M(u^*, v^*) \leq K$, we cannot conclude at this point that there are no documents satisfying the threshold. This is because there can be some documents in which most occurrences of P_1 and P_2 correspond to the fringe leaves. Hence, we check the documents corresponding to fringe leaves separately using the *Check-Fringe* operation described below. Similarly, for K-repeats, if $M(u^*, v^*) \leq K$, it will immediately imply that there exists some document d_i with $mindist_i(P_1, P_2) \leq K$. Otherwise, we cannot conclude anything yet, and we shall check the fringe leaves. The Check-Fringe operations for K-mine and K-repeats are as follows.

Check-Fringe for K-mine: Firstly we identify the document d_c corresponding to each fringe leaf ℓ_a by traversing the wavelet tree WT of D_A. Let B_1 represent the bit-vector in the root of WT. If $B_1[a] = 0$, we move to the $rank_0(B_1, a)$th position in the left child of B_1, else we move to the $rank_1(B_1, a)$th position in the right child of B_1. Here $rank_0(B, a)$ and $rank_1(B, a)$ represent the number of $0's$ and $1's$ in $B[1..a]$, respectively. This procedure is continued recursively until we reach a leaf node in WT, and the document d_c is identified. Then we can count the number of suffixes in $[L_1, R_1]$ and $[L_2, R_2]$ that belong to d_c. This can be done by traversing the WT of D_A using a similar fashion. Consequently, we obtain the total occurrences of P_1 and P_2 in the document d_c, and check if it is above the threshold K.

Check-Fringe for K-repeats: Similar to the above described method, for each fringe leaf ℓ_f, we first identify the corresponding document d_c. Next, for those suffixes that correspond to d_c within the suffix range $[L_1, R_1]$, we identify the contiguous suffix range $[l_1, r_1]$ in the individual SA of d_c. Such a *translation* operation can be done by traversing the WT of D_A (i.e. by translating $[L_1, R_1]$ to the leaf node corresponding to document d_c). Similarly, we obtain the translated suffix range $[l_2, r_2]$ for $[L_2, R_2]$.

Let $\ell_{f'}$ be the position of ℓ_f after translation and SA_c denotes the suffix array of d_c. Then we need to check the following: (Case 1) If $\ell_{f'} \in [l_1, r_1]$, we check if there exists any $\ell_{f''} \in [l_2, r_2]$ such that $|SA_c[\ell_{f'}] - SA_c[\ell_{f''}]| \leq K$. (Case 2) Otherwise, if $\ell_{f'} \in [l_2, r_2]$, we check if there exists any $\ell_{f''} \in [l_1, r_1]$ such that $|SA_c[\ell_{f'}] - SA_c[\ell_{f''}]| \leq K$. In order to perform these checks efficiently, we maintain the wavelet tree WT_c of SA_c and perform the following orthogonal range searching queries, respectively [10]:

Case 1: Use position range $[l_2, r_2]$, value bound $[SA_c[\ell_{f'}] - K, SA_c[\ell_{f'}] + K]$;
Case 2: Use position range $[l_1, r_1]$, value bound $[SA_c[\ell_{f'}] - K, SA_c[\ell_{f'}] + K]$.

If the output (number of occurrences) of this search is at least one, then d_c is a valid output for the K-repeats query[2].

Analysis: In both K-mine and K-repeats, the Check-Fringe operation involves a wavelet tree traversal per fringe leaf. Since the total number of fringe leaves is less than $4g$, the total time for the Check-Fringe operations is $O(g \log n) = O(\sqrt{n/\beta} \log n)$ [10].

Lemma 4. *We can maintain an $O(n\beta)$-space Any-One index, such that given a query consisting two patterns P_1 and P_2, and a parameter K, we can answer if the number of outputs for the K-mine or the K-repeats queries is at least one, using $O(|P_1| + |P_2| + \sqrt{n/\beta} \log n)$ time.* □

Using an Any-One index, top-1 queries can be answered with the same time bounds. In the next section, we describe our main indexes for answering the original K-mine and K-repeats queries.

4 Efficient Indexes for K-Mine and K-Repeats Problems

We maintain the collection of documents $\mathcal{D}$ in the form of a balanced binary tree, such that the root node represents all documents and further these documents are divided evenly among the child nodes. This procedure is continued recursively until we reach a node (leaf) which represents a single document. We maintain an Any-One index for each $\mathcal{D}_k$, where $\mathcal{D}_k$ is the set of documents represented by an internal node W_k in the binary tree. Query answering is performed as follows: first we check if there exists at least one document in the whole set $\mathcal{D}$, which satisfies our threshold condition. This is performed using the Any-One index at the root of the binary tree. If the answer is YES, we do a multi-way search in both child nodes, which are two mutually exclusive and exhaustive subsets of $\mathcal{D}$. This procedure is continued recursively until we reach a leaf node in the binary tree. At any node, if the Any-One index returns NO, we do not need to continue the recursive step further in its sub-tree.

To maintain the Any-One index for $\mathcal{D}_k$, we will need to maintain the wavelet tree WT_k of the subsequence of the document array whose documents belong to $\mathcal{D}_k$. Instead of storing these wavelet trees separately for each node, we can just store the wavelet tree WT of the original D_A. It is because WT_k is exactly the same as the sub-tree of WT rooted at W_k. Next, let GST_k denote the Generalised suffix tree for $\mathcal{D}_k$, which is a sub-graph of the original GST for $\mathcal{D}$. Therefore, instead of storing all GST_k's, we store only the original GST and the parenthesis encoding [16,1,13] (along with the marked nodes information) of individual GST_k's.

To speed up the query, we hope that the searching of P_1 and P_2 in GST_k can be avoided. That is, when we query the Any-One index for $\mathcal{D}_k$, we hope that the desired marked nodes and the ranges, can be obtained without searching

[2] Here, we shall use the range successor query [21] in the case of K-repeats.

P_1 and P_2 again. This can be performed by translating the suffix ranges using the wavelet-tree (to the node W_k in WT). Let $[L_{1k}, R_{1k}]$ and $[L_{2k}, R_{2k}]$ be the translated suffix ranges in GST_k, then the corresponding marked nodes can be obtained in $O(1)$ (as described in Lemma 3) using parenthesis encoding and marked nodes information.

However, there is a challenge involved. We will need to maintain a score matrix separately for each Any-One index. The total space thus exceeds $O(n\beta)$, so that we will need to use a larger group size (roughly $\sqrt{\log n}$ times) to reduce the total space back to $O(n\beta)$. Consequently, the time spent at any node W_k for Check-Fringe operation could be $O(\sqrt{n_k/\beta}\log^{1.5} n)$, and in the worst case, this could lead to a total of $O(\sqrt{n_k/\beta}\log^{2.5} n)$ time. Note that the $\log n$ blow-up in the worst case comes from the fact that n_k may remain nearly the same if we go down one level in the wavelet tree. In order to achieve faster searching, we use the weight-balanced wavelet tree (WBT) instead, so that we can ensure that n_k is getting reduced exponentially as we traverse down in the tree. Further, we choose a separate blocking factor $g_k = 2^{-\delta_k/2}\sqrt{(n\log n/\beta)}$ for each bit-vector B_k at the node W_k with depth δ_k. In summary, our index consists of the following components.

1. GST: $O(n)$ space;
2. WBT: $O(\sum |d_i| \log(n/|d_i|) + n) + o(n \log |\mathcal{D}|) = O(n \log |\mathcal{D}|)$ bits of space;
3. Parenthesis encodings of GST_k's: It consists of the encodings for GST_k ($4n_k$ bits: suffix tree of a text of length n_k contains at the max $2n_k - 1$ nodes) and for the marked nodes E_k ($2n_k$ bits). Hence the total space is $O(\sum_k n_k) =$ $O(\text{size of } WBT) = O(n \log |\mathcal{D}|)$ bits;
4. Score matrices: The space for score matrix M_k associated with the Any-One index at W_k is $O(n_k^2/g_k^2)$. By Lemma 1, $n_k \leq 4n/2^{\delta_k}$. Now, since g_k is set to $2^{-\delta_k/2}\sqrt{(n\log n/\beta)}$, M_k takes $O(2^{-\delta_k} n\beta/\log n)$ space. The number of nodes at a depth i is almost 2^i. So the total space taken by all score matrices can be bounded as $O(\sum_k (n_k/g_k)^2) = \sum_{i=0}^{\log n} 2^i (2^{-i} n\beta/\log n) = O(n\beta)$.

Lemma 5. *The total space for the above index is $O(n\beta)$ words.* □

4.1 Query Answering

The query consists of input patterns P_1 and P_2 and a parameter K. First we search for these patterns in GST and find their corresponding suffix ranges. Once we get these ranges, we do not need to use the GST any more. Starting from the root node in the weight-balanced wavelet tree, we use the parenthesis encodings to find the marked ancestors u^* and v^*, and check the score matrix entry. If the score matrix entry satisfies the threshold condition (i.e., $M(u^*, v^*) = \max_i\{f_{iu^*} + f_{iv^*}\} \geq K$ for K-mine, and $M(u^*, v^*) = \min_i\{mindist_i(P_1^*, P_2^*)\} \leq K$ for K-repeats), we translate both these ranges into the child nodes of the WBT and continue the procedure recursively until we reach a node which represents a single document. This document is listed as a valid output. During this WBT traversal, whenever the threshold condition is not satisfied at some node, we do

Check-Fringe operations and we will not traverse further down in its sub-tree. In summary, we have the following theorem.

Theorem 1. *We can design an $O(n\beta)$ space index for answering K-mine or K-repeats queries (on two patterns) in $O(|P_1| + |P_2| + t \log n + \sqrt{(nt/\beta)} \log^2 n)$ time, where t is the number of outputs and β is a tunable parameter.*

Proof sketch: The query time mainly consists of the time for tree traversal, the time for the Check-Fringe operations, and the $O(|P_1| + |P_2|)$ time for the initial search in GST. For any leaf node which produces a valid output, we may have checked its sibling node (which can be a leaf), so that the number of leaf nodes visited can be at the most $2t$. Since the height of the tree is $O(\log n)$, the total number of nodes visited (tree traversal time) is $O(t \log n)$. During the tree traversal, if we perform *Check-Fringe* operation at a node, we do not traverse further down from that node. Hence the number of such nodes where we perform *Check-Fringe* operations can be bounded by $O(t \log n)$. Here if we ignore all the nodes in WBT which were not visited, it can be viewed as a binary tree Δ such that all the nodes in WBT where we performed *Check-Fringe* operations become a leaf node in Δ. The number of *Check-Fringe* operations at a leaf ℓ (in Δ at a depth $depth(\ell)$) is $O(2^{-depth(\ell)/2}\sqrt{n \log n/\beta})$ (which is the blocking factor of the GST_k corresponding to that node). Hence the total number for *Check-Fringe* operations can be bounded as $O(\sum_\ell 2^{-depth(\ell)/2}\sqrt{n \log n/\beta})$. We have $\sum_\ell 2^{-depth(\ell)/2} \leq \sqrt{(\sum_\ell 1^2)(\sum_\ell 2^{-depth(\ell)})}$ using Cauchy-Schwarz's inequality[3] and for any binary tree $\sum_\ell 2^{-depth(\ell)} \leq 1$ according to Kraft's inequality. In our case $\sum_\ell 1^2 = O(t \log n)$ and time per *Check-Fringe* operation is $O(\log n)$. Therefore the total time for *Check-Fringe* operations can be bounded by $O(\sqrt{(nt/\beta)} \log^2 n)$. □

In the document listing problem the score is just a binary (YES or NO) and we do not store the document id, hence we choose a smaller blocking factor ($g_k = 2^{-\delta_k/2}\sqrt{n/\beta}$) which improves the query time to $O(|P_1| + |P_2| + t \log n + \sqrt{(nt/\beta)} \log^{3/2} n)$.

5 Top-K Queries

For the sake of similicity, assume all top-K answers are coming from leaf nodes(ignore fringe leaves for now) in WBT. Those K leaves can be found out in $O(K \log^2 n)$ time as follows. First we find the leaf (say ℓ^1)corresponding to the top-1 score, by greedly following the branch with highest score matrix entry. Note that we travel $O(\log n)$ nodes to reach this leaf node. In order to find the leaf(ℓ^2) corresponding to top-2, we *insert* all the nodes (along with the score), which are the siblings of nodes in the path(ℓ^1) into a *max-heap*. Then we do an *extract-max* operation to find the top-2 score and we traverse further down from

[3] $\sum_{i=1}^n x_i y_i \leq \sqrt{\sum_{i=1}^n x_i^2}\sqrt{\sum_{i=1}^n y_i^2}$.

that sibling node greedily to reach ℓ^2. We do this procedure K times (which includes $O(K \log n)$ *insert* and K *extract-max* operations). Hence the total time is $O(K \log^2 n)$.

Now we need to check the fringe leaves which we have ignored before. Using similar analysis as in section 5, this can be performed in $O(\sqrt{nK/\beta} \log^2 n)$ time. For retrieving the top-K answers, we sort all the answers from fringe leaves and the K answers from leaf nodes together and retrieve the Top-K highest scored (unique) documents.

Theorem 2. *We can design an $O(n\beta)$ space index for answering top-K version of the document retrieval queries (on two patterns) in $O(|P_1| + |P_2| + K \log^2 n + \sqrt{(nK/\beta)} \log^2 n)$ time, where β is a tunable parameter.* □

6 Generalization to Multi-pattern Queries

Finally, we show how to extend our indexes to handle multi-pattern queries where query consists of m patterns $P_1, P_2, ..., P_m$. Similar to our two-pattern index, we maintain a generalized suffix tree of all documents and a wavelet tree over the document array. Since the score of a particular document depends upon m patterns, we use an m-dimensional score matrix. In order to maintain space bounds we choose the blocking factor $g_k = \tilde{O}(2^{-\delta_k(1-1/m)} n^{1-1/m} \beta^{-1/m})$. Therefore the number of marked nodes corresponding to GST_k is $O(n_k/g_k) = \tilde{O}(2^{-\delta_k} n\beta)^{1/m}$ and the size of the score matrix $M_k = \tilde{O}((n/g_k)^m) = \tilde{O}(2^{-\delta_k} n\beta)$. By carrying out similar analysis as before (except we use Hölder's inequality[4] instead of Cauchy-Schwarz's inequality), we obtain the following Theorem.

Theorem 3. *For multi-pattern queries, document listing, K-mine, and K-repeats problems can be answered in $O(\sum_{i=1}^{m} |P_i|) + \tilde{O}(t + (t/\beta)^{1/m} n^{1-1/m})$ and Top-K queries in $O(\sum_{i=1}^{m} |P_i|) + \tilde{O}(K + (K/\beta)^{1/m} n^{1-1/m})$ time by maintaining an $O(n\beta)$-space index.* □

Note that the space requirement of our index can be reduced by using a Compressed Suffix Array [7,5] instead of GST and by tuning the score matrix size to $o(n)$. It still remains an open question if a fully succinct space bound (i.e., without $O(n \log |\mathcal{D}|)$ bits term of WBT size) can be achieved for these problems.

References

1. Bender, M.A., Farach-Colton, M.: The LCA Problem Revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
2. Brin, S., Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks 30(1-7), 107–117 (1998)
3. Cohen, H., Porat, E.: Fast Set Intersection and Two Patterns Matching. In: LATIN (2010)

[4] Hölder's inequality: $\sum_{i=1}^{n} x_i y_i \leq (\sum_{i=1}^{n} x_i^p)^{1/p} (\sum_{i=1}^{n} y_i^q)^{1/q}$, where $1/p + 1/q = 1$.

4. Ferragina, P., Giancarlo, R., Manzini, G.: The Myriad Virtues of Wavelet Trees. Inf. and Comp. 207(8), 849–866 (2009)
5. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms (TALG) 3(2) (2007)
6. Grossi, R., Gupta, A., Vitter, J.S.: High-Order Entropy-Compressed Text Indexes. In: SODA, pp. 841–850 (2003)
7. Grossi, R., Vitter, J.S.: Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. SICOMP 35(2), 378–407 (2005)
8. Hon, W.K., Shah, R., Vitter, J.S.: Ordered Pattern Matching: Towards Full-Text Retrieval. Tech Report TR-06-008, Dept. of CS, Purdue University (2006)
9. Hon, W.K., Shah, R., Vitter, J.S.: Space-Efficient Framework for Top-k String Retrival Problems. In: FOCS, pp. 713–722 (2009)
10. Mäkinen, V., Navarro, G.: Rank and Selected Revisited and Extended. TCS 387(3), 332–347 (2007)
11. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SICOMP 22(5), 935–948 (1993)
12. Matias, Y., Muthukrishnan, S., Sahinalp, S.C., Ziv, J.: Augmenting Suffix Trees, with Applications. In: Bilardi, G., Pietracaprina, A., Italiano, G.F., Pucci, G. (eds.) ESA 1998. LNCS, vol. 1461, pp. 67–78. Springer, Heidelberg (1998)
13. Munro, J.I., Raman, V.: Succinct Representation of Balanced Parentheses and Static Trees. SICOMP 31(3), 762–776 (2001)
14. Muthukrishnan, S.: Efficient Algorithms for Document Retrieval Problems. In: SODA, pp. 657–666 (2002)
15. Raman, R., Raman, V., Rao, S.S.: Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. TALG 3(4) (2007)
16. Sadakane, K.: Compressed Suffix Trees with Full Functionality. TCS, 589–607 (2007)
17. Sadakane, K.: Succinct Data Structures for Flexible Text Retrieval Systems. JDA 5(1), 12–22 (2007)
18. Välimäki, N., Mäkinen, V.: Space-Efficient Algorithms for Document Retrieval. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 205–215. Springer, Heidelberg (2007)
19. Weiner, P.: Linear Pattern Matching Algorithms. In: Proc. Switching and Automata Theory, pp. 1–11 (1973)
20. Wu, S.B., Hon, W.K., Shah, R.: Efficient Index for Retrieving Top-k Most Frequent Documents. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 182–193. Springer, Heidelberg (2009)
21. Yu, C.C., Hon, W.K., Wang, B.F.: Efficient Data Structures for the Orthogonal Range Successor Problem. In: Ngo, H.Q. (ed.) COCOON 2009. LNCS, vol. 5609, pp. 96–105. Springer, Heidelberg (2009)