Journal of Discrete Algorithms 8 (2010) 402–417

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Efficient index for retrieving top-k most frequent documents [☆]

Wing-Kai Hon [a,*], Manish Patil [b], Rahul Shah [b], Shih-Bin Wu [a]

[a] *Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan*
[b] *Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA*

ARTICLE INFO

Article history:
Received 8 July 2009
Accepted 19 August 2010
Available online 15 September 2010

Keywords:
Document retrieval
Text indexing
Induced generalized suffix tree

ABSTRACT

In the *document retrieval problem* (Muthukrishnan, 2002), we are given a collection of documents (strings) of total length D in advance, and our target is to create an index for these documents such that for any subsequent input pattern P, we can identify which documents in the collection contain P. In this paper, we study a natural extension to the above document retrieval problem. We call this *top-k frequent document retrieval*, where instead of listing all documents containing P, our focus is to identify the top-k documents having most occurrences of P. This problem forms a basis for search engine tasks of retrieving documents ranked with TFIDF (Term Frequency-Inverse Document Frequency) metric.

A related problem was studied by Muthukrishnan (2002) where the emphasis was on retrieving all the documents whose number of occurrences of the pattern P exceeds some frequency threshold f. However, from the information retrieval point of view, it is hard for a user to specify such a threshold value f and have a sense of how many documents will be reported as the output. We develop some additional building blocks which help the user overcome this limitation. These are used to derive an efficient index for top-k frequent document retrieval problem, answering queries in $O(|P| + \log D \log\log D + k)$ time and taking $O(D \log D)$ space. Our approach is based on a new use of the suffix tree called *induced generalized suffix tree* (IGST). The practicality of the proposed index is validated by the experimental results.

1. Introduction

String matching problems have been studied for more than three decades [10,3,9]. In the simplest form, we are given a relatively long character string, called *text*, and a relatively short character string, called *pattern*, in which our target is to locate all occurrences of the pattern within the text. In some applications, the text is given in advance, and we may preprocess it and create an auxiliary data structure – called an *index* – for the text, so that any subsequent pattern matching query can be answered more efficiently. For instance, if *suffix tree* [13,19] – a linear-space index – for the text is created, locating all *occ* occurrences of a pattern P of length $|P|$ can be done in *optimal* $O(|P| + occ)$ time, irrespective of the length of the text.

In string databases or in string retrieval systems, we have a collection Δ of multiple documents (strings) instead of just one text string. In this case, the basic problem is to retrieve all the documents in which the query pattern P occurs.

☆ A preliminary version appears in *Proceedings of Symposium on String Processing and Information Retrieval*, 2009. This work is supported in part by Taiwan NSC Grants 96-2221-E-007-082-MY3 and 99-2221-E-007-123-MY3 (Wing-Kai Hon), and by US NSF Grants CCF-0621457 and CCF-1017623 (Rahul Shah).

* Corresponding author. Postal address: 101 Kuang Fu Road (Section 2), Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan 300, ROC. Tel.: +886 35731307; fax: +886 3572 3964.

E-mail addresses: wkhon@cs.nthu.edu.tw (W.-K. Hon), mpatil@csc.lsu.edu (M. Patil), rahul@csc.lsu.edu (R. Shah), hel@alumni.nccu.edu.tw (S.-B. Wu).

1570-8667/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2010.08.003

This is known as *document retrieval* problem and has been studied by Matias et al. [12] and Muthukrishnan [14]. The main issue here is that there may be many occurrences of the pattern over the entire collection Δ, but the overall number of documents in which the pattern occurs might be much smaller. Thus, the naive method of finding all the occurrences first and then reporting unique documents is far from efficient. Muthukrishnan [14] gave an optimal $O(D)$-space data structure which answers the document retrieval query in $O(|P| + output)$, where *output* is the number of documents which contain the pattern P. This has been a popular approach of many subsequent papers [15,18] which attempted to derive succinct/compressed data structures for this problem.

A more interesting variant was also proposed in [14]. In this variant we need to retrieve only those documents which have more than f occurrences of the pattern; we call this the f-*mine* problem.[1] In terms of information retrieval this is a more interesting query because it attempts to obtain only those documents which are highly relevant. The notion of relevance here is simply the term frequency. Sadakane [15] also gave a method to compute TFIDF (Term Frequency-Inverse Document Frequency) scores of each retrieved document. However, what is lacking here is the notion of top-k documents with the highest TFIDF scores. In [15] one needs to retrieve all the documents first, and then only the scores are computed. Related to document retrieval, a more general problem of position-restricted substring matching was introduced by [11,7]. Also, the study of top-k indexing and rank sensitive data structures was carried by [2]. None of these results are directly applicable to the problem studied in this paper.

The problem considered in this paper is closely related to the f-mine problem. In our case, we directly find top-k documents having the maximum number of occurrences for the given pattern. We build new primitives like *inverse document mine* query, which quickly allows us to find a threshold f_k (or simply f when the context is clear) which is the frequency of the pattern in kth frequent document. Based on this we can draw strong connections with the f-mine problem. The main component of our solution is called *induced generalized suffix tree* (IGST), which is structurally the same as the index proposed by [14]. However, we show a new application to the IGST where we "linearize" it and combined it with successor searching functionality to obtain the desired performance.

For the sake of completeness we mention here that if theoretical performance guarantees are not the main concern, then such problems are practically solved using inverted indexes [22]. However, inverted indexes either only allow efficient searching for certain predefined pattern, or they take a lot more space (if they were to answer for arbitrary patterns). Our solution provides theoretical guarantees but can also be seen as a modification of inverted index where the lists for the patterns (which are contained within some other patterns) are smartly combined to reduce space.

1.1. Our problems

In this paper, we study two natural extensions to the above document retrieval problems. The first one is called the *inverse document mining* problem, which is defined as follows:

Problem 1 *(Inverse document mining problem).*

> *Given*: A collection Δ of documents, with total length D;
> *Target*: Create an index for Δ to support the following query efficiently:
> On given any input pattern P and any input integer k, find the frequency f such that $\rho_{f+1} < k \leqslant \rho_f$, where ρ_f denotes the number of documents in Δ containing at least f occurrences of P. In other words, we want to find the largest f such that there are k documents with f occurrences of P.
> We denote the above query by *inverse_mine*(P, k).

Example. Suppose there are five documents, T_1, T_2, T_3, T_4, and T_5, in the set Δ. Also, the number of times a pattern P occurring in these documents are $15, 24, 3, 3, 1$, respectively. On the query *inverse_mine*$(P, 2)$, we should return $f = 15$.

The second extension is called the *top-k document retrieval* problem, which is motivated by the need of finding the "most relevant" documents in the output of a document retrieval query. Here, we assume that the relevance of a document with respect to a pattern P is measured by the number of times P occurring in the document. Our problem is then defined as follows:

Problem 2 *(Top-k document retrieval problem).*

> *Given*: A collection Δ of documents, with total length D;
> *Target*: Create an index for Δ to support the following query efficiently:
> On given any input pattern P and any input integer k, find the k documents in Δ which contain the most occurrences of P. We assume tie is broken arbitrarily in case two documents contain the same number of P.
> We denote the above query by *top_document*(P, k).

[1] In [14], this problem is called as K-mine, where K denotes the frequency threshold.

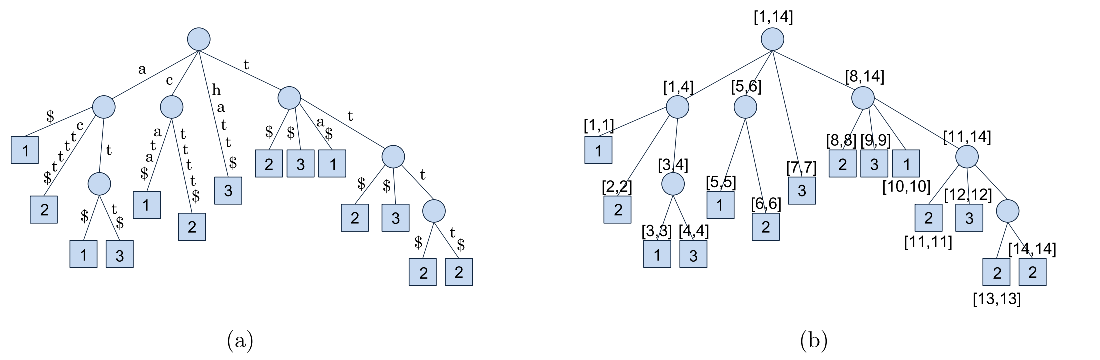

Fig. 1. (a) Example of a GST. The set Δ contains three documents, which are $T_1 = \texttt{cata}$, $T_2 = \texttt{actttt}$, and $T_3 = \texttt{hatt}$. The label in each leaf (square-shaped node) denotes the origin of the corresponding suffix. (b) Suffix ranges in the GST. Above or below each node shows its suffix range.

Example. Suppose there are five documents, T_1, T_2, T_3, T_4, and T_5, in the set Δ. Also, the number of times a pattern P occurring in these documents are 15, 24, 3, 3, 1, respectively. On the query *top_document*$(P, 2)$, we should return $\{T_1, T_2\}$. However, on the query *top_document*$(P, 3)$, we may return either $\{T_1, T_2, T_3\}$ or $\{T_1, T_2, T_4\}$, as tie is broken arbitrarily among documents with the same number of occurrences of P.

By adapting Muthukrishnan's $O(D \log D)$-space indexes [14], the queries in the above two problems can readily be supported in $O(|P| \log D)$ time and $O(|P| \log D + k)$ time, respectively. In this paper, we propose alternative indexes with the same space, so that the queries are supported in $O(|P| + \log D \log\log D)$ time and $O(|P| + \log D \log\log D + k)$ time, respectively. These indexes thus outperform the naive extension of Muthukrishnan's index whenever P is sufficiently long; precisely, when $|P| = \omega(\log\log D)$.

The core of our indexes, called the *induced generalized suffix trees*, are structurally equivalent to the core of the indexes proposed in [14]; the major difference lies in the information being stored. Consequently, we are able to support the new types of query, and alter the searching methods to obtain the desired trade-off in query times.

1.2. Paper organization

The remainder of the paper is as follows. Section 2 introduces two basic tools which form the building blocks of our indexes. Section 3 describes the induced generalized suffix trees (IGST), with which we can construct the index for the inverse document mining problem. In Section 4, we show how to adapt the IGST slightly to solve the top-k document retrieval problem. Section 5 discusses how to construct our index efficiently. Section 6 gives the empirical studies on the practicality of our index. We conclude in Section 7 with some open problems.

2. Basic tools

2.1. Generalized suffix tree

Let $\Delta = \{T_1, T_2, \ldots, T_m\}$ denote a set of documents. Each document is a character string with characters drawn from a common alphabet Σ whose size $|\Sigma|$ can be unbounded. For notation purpose, we assume that for each i, the last character of document T_i is marked by a special character $\$_i$, which is unique among all characters in all documents.[2] The *generalized suffix tree* [13,19] (GST) for Δ is a compact trie storing all suffixes of each T_i. Precisely, each suffix of each document corresponds to a distinct leaf in the GST. Each edge is labeled by a sequence of characters, such that for each leaf representing some suffix s, the concatenation of the edge labels along the root-to-leaf path is exactly s. In addition, for any internal node u, the edges incident to its children all differ by the first character in the corresponding edge labels, so that the children of u are ordered according to the alphabetical order of such a first character. See Fig. 1(a) for an example.

The following property of the GST is immediate:

Lemma 1. *Consider all suffixes of all documents in Δ. The jth smallest suffix corresponds to the jth leftmost leaf in GST.*

Next, we define an important concept called *suffix range*:

[2] When the context is clear, we shall simply denote each $\$_i$ by the same character \$.

Definition 1. Consider the subtree of a node u in the GST. Let v and w be the leftmost and the rightmost leaves in this subtree. Furthermore, let ℓ and r denote the rank of v and the rank of w among all leaves in the GST, respectively; precisely, v is the ℓth leftmost leaf and w is the rth leftmost leaf in GST. Then, the range $[\ell, r]$ is called the *suffix range* of u.

Fig. 1(b) gives an example of the suffix range. A simple observation is shown as follows:

Lemma 2. *Let u and v be two nodes in the GST, and let $[\ell_u, r_u]$ and $[\ell_v, r_v]$ be their suffix ranges, respectively. The two ranges are disjoint if and only if there is no ancestral-descendant relationship between u and v. In case u is an ancestor of v, we have $\ell_u \leqslant \ell_v \leqslant r_v \leqslant r_u$.*

For each node v, we use *path*(v) to denote the concatenation of edge labels along the path from root to v. Then, we define the concept of *locus* as follows:

Definition 2. For any string Q, the *locus* of Q in the GST is defined to be the highest node v (i.e., nearest to the root) such that Q is a prefix of *path*(v). In case no v satisfies the condition, the locus of Q is *null*.

Example. Consider the GST in Fig. 1(a). The locus of the string `ttt` is the parent node of the two rightmost leaves, while the locus of the string `hat` is the 7th leftmost leaf. The locus of the string `cap` is null, as there is no node v such that `cap` is a prefix of *path*(v).

Note that the locus of Q can be determined in $O(|Q|)$ time by traversing the GST and matching characters of Q from the beginning to the end.

It is easy to see that if a pattern P occurs at position j in a text T, P must be a prefix of the suffix of T which starts at position j. The converse is also true. Based on this, we have the following lemma which captures the power of GST in pattern matching:

Lemma 3. *A pattern P occurs in some document of Δ if and only if the locus of P is not null. In addition, each leaf in the subtree rooted at the locus of P corresponds to a distinct occurrence of P, and vice versa.*

2.2. Optimal index for colored range query

Let $A[1..n]$ be an array of length n, with each entry storing a color drawn from $C = \{1, 2, \ldots, c\}$. A *colored range query*, denoted by $CRQ(i, j)$, receives two input integers i and j with $1 \leqslant i \leqslant j \leqslant n$, and outputs the set of colors contained in the subarray $A[i..j]$. For instance, suppose A has seven entries, which are colored by 1, 3, 2, 6, 2, 4, and 5, respectively. Then, the query $CRQ(2, 5)$ requests the set of colors in the subarray $A[2..5]$, which should return $\{2, 3, 6\}$.

An index is proposed in [14] for answering colored range query in an *output-sensitive* manner; the performance of the index is summarized in the following lemma:

Lemma 4. *An index for the array A can be created using $O(n)$-space of storage, such that for any i and j, the colored range query $CRQ(i, j)$ can be answered in $O(\gamma)$ time, where γ denotes the number of (distinct) colors in the output set.*

2.3. Y-Fast trie for efficient successor query

Let S be a set of n distinct integers taken from $[1, D]$. Given an input x, a *successor query* on S reports the smallest integer in S that is greater than or equal to x. An efficient index for this query, called *y-fast trie*, was proposed by [21], whose performance is summarized as follows:

Lemma 5. *An index for the set S of n integers can be created using $O(n)$-space of storage, such that for any input x, the successor query $succ(S, x)$ can be answered in $O(\log\log D)$ time, where D denotes the universe where integers of S are chosen from. The index can be constructed in randomized $O(n \log D)$ time.*

3. Induced generalized suffix tree

This section defines the *induced generalized suffix tree for frequency f*, or *IGST-f*, which can be used to count the number of documents with P occurring at least f times. Then, we give an array representation of IGST-f, and show how to support *inverse_mine*(P, k) query efficiently.

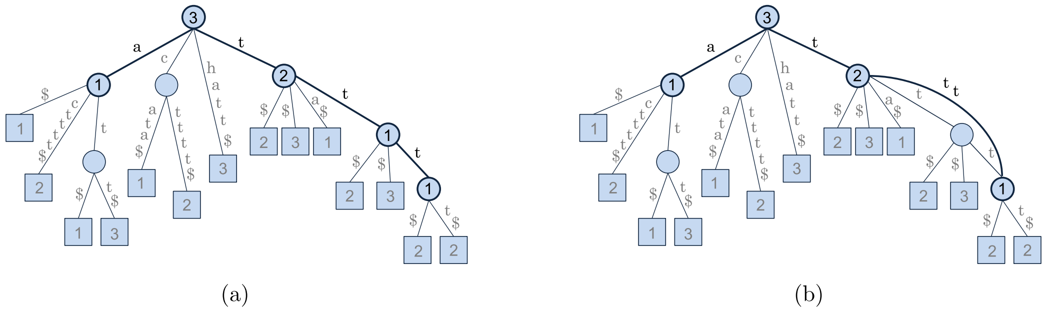

Fig. 2. (a) Example of pre-IGST-2. (b) Example of IGST-2.

3.1. IGST-f: The IGST for frequency f

First, we define a tree induced from the GST, called *pre-IGST-f*, as follows:

Definition 3. Consider the GST for Δ and an integer f with $1 \leqslant f \leqslant D$. Suppose each leaf of GST is labeled by the origin (document) of the corresponding suffix. For each internal node v of the GST, we say v is *f-frequent* if in the subtree rooted at v, at least f leaves have the same label. The induced subtree of GST formed by retaining all f-frequent nodes is called the *pre-IGST-f*.

Example. In Fig. 2(a), all 2-frequent internal nodes in the GST of Fig. 1(a) are highlighted, and the induced subtree formed by these highlighted nodes is the pre-IGST-f.

Definition 4. Let v be a node in the pre-IGST-f, so that v is an internal node in the GST. We use *count*(f, v) to denote the number of distinct document with at least f leaves labeled by it in the subtree rooted at v in the GST. We simply use *count*(v) instead of *count*(f, v) when context is clear.

Example. In the pre-IGST-2 shown in Fig. 2(a), each internal node v is labeled by the corresponding *count*(v). Note that *count*(v) must be between 1 and m, where m denotes the number of documents in Δ.

The following two lemmas demonstrate the pattern matching power of pre-IGST-f, which can both be proved easily based on Lemma 3:

Lemma 6. *A pattern P occurs at least f times in some document of Δ if and only if the locus of P in pre-IGST-f is not null.*

Lemma 7. *Let ρ_f denote the number of documents in Δ with pattern P occurring at least f times. If the locus of P in pre-IGST-f is null, then $\rho_f = 0$; otherwise, $\rho_f = count(v)$, where v is the locus of P.*

Next, we describe IGST-f, which is in fact a simplified version of pre-IGST-f.

Definition 5. Consider the pre-IGST-f. For each internal node v, we say v is *redundant* if (i) v is a degree-1 node and (ii) *count*$(v) =$ *count*(child(v)), where child(v) denotes the unique child of v. The induced tree formed by contracting all redundant nodes in the pre-IGST-f is called the *IGST-f*.

Example. In the pre-IGST-2 shown in Fig. 2(a), the locus of the string `tt` is a *redundant* internal node because it is of degree-1 and it has the same *count* as its child. On the other hand, the locus of the string `t` is not redundant, despite it is of degree-1. By contracting all redundant nodes, we obtain IGST-2 as shown in Fig. 2(b).

Observe that when a node v in the pre-IGST-f is redundant, the set of the *count*(v) documents corresponding to v (where each of them has at least f labels in the subtree rooted at v in the GST) is exactly the same as the set of the *count*(child(v)) documents corresponding to child(v), so that the two counts are the same. This observation immediately leads to the following lemma:

Lemma 8. *The locus of a pattern P in pre-IGST-f is not null if and only if the locus of P in IGST-f is not null. In case the locus is not null, let v and w denote the locus of P in pre-IGST-f and the locus of P in IGST-f, respectively. Then, count$(v) =$ count(w).*

The structure of our IGST-f is equivalent to the structure of the index proposed by Muthukrishnan [14] to solve the *document mining* problem, which on given a frequency f and a pattern P, reports all $\mathtt{occ}_f$ documents which have at least f occurrences of P in $O(|P| + \mathtt{occ}_f)$ time. By using this structure, we can thus readily answer *inverse_mine*(P, k) query by binary search on f, where each step in the binary search checks if the specific f is the desired answer for *inverse_mine*(P, k). Note that each binary search step only needs to confirm if $k \geqslant \mathtt{occ}_f$, so that it takes $O(|P| + k)$ time. In total, the desired answer can be obtained in $O((|P| + k)\log D)$ time.

Furthermore, if we assume that the value of *count*(v) is stored for each node v in the IGST-f (which is not required in [14]), we obtain the following theorem:

Theorem 1 *(Adapted index of Muthukrishnan). By storing all IGST-1, IGST-2, ..., and IGST-D, we can answer inverse_mine(P, k) query, for any input pattern P and input integer k, in $O(|P|\log D)$ time.*

Proof. For any f, we can find the locus of P in IGST-f in $O(|P|)$ time, analogous to finding locus in the GST. Then, we can determine the number of documents containing at least f occurrences of P, based on Lemma 6. To answer *inverse_mine*(P, k), it is sufficient to search for $O(\log D)$ IGSTs, based on a binary search of f, thus using $O(|P|\log D)$ time in total. □

3.2. Array representation of IGST-f

In Theorem 1, answering the *inverse_mine* query requires finding the locus of P in each IGST during the binary search. This could be time-consuming when P is long. In the following, we propose a simple alternative scheme that allows each locus-finding step to be done in $O(\log D)$ time instead of $O(|P|)$ time, thus giving a trade-off in the query time.

Our scheme is to make use of the suffix ranges. Firstly, suppose that the suffix range of the locus of P is already computed. Let $[\ell_P, r_P]$ denote this range if the locus exists. Next, suppose each node in IGST-f is associated with the suffix range of the corresponding node in the GST. Then, we have the following observation:

Lemma 9. *The locus of P in IGST-f, if exists, is the node v such that* (i) *the associated suffix range of any descendant of v (including v) is a subrange of $[\ell_P, r_P]$, and* (ii) *the associated suffix range of its parent node is not a subrange of $[\ell_P, r_P]$.*

Proof. By definition, P is a prefix of *path*(v), so that by Lemma 2 and by the definition of IGST, the associated suffix range of v, and also any of its descendant, must be a subrange of $[\ell_P, r_P]$. On the other hand, the associated suffix range of the parent of v must not be a subrange of $[\ell_P, r_P]$, since otherwise, v is not the node nearest to the root having P as a prefix of *path*(v), contradicting the definition of locus. □

Next, consider performing a pre-order traversal on the IGST-f, and enumerating the associated suffix range of a node as it is visited. Let $[\ell(z), r(z)]$ denote the suffix range of the zth node enumerated during the traversal. Now, suppose that the locus of P in IGST-f (assuming exists) is the jth node in the pre-order traversal of IGST-f. That is, the locus of P in IGST-f has associated suffix range $[\ell(j), r(j)]$. Further, suppose that we examine $[\ell(z), r(z)]$ for some z. Recall that $[\ell_P, r_P]$ denote the suffix range of the locus of P in the GST. The theorem below is the heart of our proposed index, which gives a simple way to determine the relationship between j and z:

Lemma 10. *The following statements are true, and cover all possible relationship between ℓ_P, r_P, $\ell(z)$, and $r(z)$:*

1. *if $r_P < \ell(z)$, then $j < z$;*
2. *if $\ell_P > r(z)$, then $j > z$;*
3. *if $[\ell(z), r(z)]$ is a subrange of $[\ell_P, r_P]$, then $j \leqslant z$;*
4. *if $[\ell_P, r_P]$ is a subrange of $[\ell(z), r(z)]$, then $j \geqslant z$.*

Proof. All the four statements can be proven based on Lemma 8. For Statement 1, if $r_P < \ell(z)$, the associated suffix range of the zth node, and all nodes visited after the zth node in the pre-order traversal, must be disjoint with $[\ell_P, r_P]$, so that none of them can be the locus of P. Thus, the desired locus must be visited earlier, so that $j < z$. Similarly, for Statement 2, if $\ell_P > r(z)$, then $j > z$. For Statement 3, the desired locus must be an ancestor of the zth visited node, so that it is either the zth visited node itself, or a node visited earlier in the traversal. This implies $j \leqslant z$. Similarly, for Statement 4, the desired locus must be a descendant of the zth visited node, so that $j \geqslant z$. □

Let $c(z)$ denote the *count* value of the zth node visited during the pre-order traversal of IGST-f. Instead of storing the IGST-f as a tree structure in Theorem 1, we represent it by an array I, such that the zth entry of I, $I[z]$, stores the 3-tuple $(\ell(z), r(z), c(z))$.

For instance, the I-array of IGST-2 in Fig. 2(b) is as follows:

k	1	2	3	4
I	$(1, 14, 3)$	$(1, 4, 1)$	$(8, 14, 2)$	$(13, 14, 1)$

Based on the previous theorem, we can obtain the value j, using *binary search* on the array I, such that $[\ell(j), r(j)]$ is the associated suffix range of the locus of P. Then, the value $c(j)$ thus stores the number of documents with at least f occurrences of P. (Note that if the locus of P in IGST-f does not exist, there is no z such that $[\ell(z), r(z)]$ is a subrange of $[\ell_P, r_P]$; consequently the binary search will correctly detect this.) Since the number of nodes in IGST-f is $O(D)$, the array I is of length $O(D)$, so that the binary search takes $O(\log D)$ time. This gives the following theorem:

Theorem 2. *By storing the GST, and the I-arrays for all IGST-1, IGST-2, ..., and IGST-D, inverse_mine(P, k) query can be answered, for any input pattern P and input integer k, in $O(|P| + \log^2 D)$ time.*

Proof. The GST is used to compute $[\ell_P, r_P]$ in $O(|P|)$ time. Then, we can determine the number of documents containing at least f occurrences of P in $O(\log D)$ time, by binary searching the I-array of IGST-f. To answer *inverse_mine*(P, k), it is sufficient to search for $O(\log D)$ I arrays of the IGSTs. The total time is thus $O(|P| + \log^2 D)$. □

Indeed, we can further speed up the query time by replacing each binary search in the IGST arrays with a single successor query in a slightly modified array. Consequently, the time spent in each visited IGST is reduced from $O(\log D)$ to $O(\log\log D)$ time. The idea is as follows. First, we observe that each node in the IGST has a natural correspondence in the original GST; precisely, a node u with suffix range $[\ell, r]$ exists in the IGST implies that a node u' with the same suffix range exists in the original GST. Next, we perform a pre-order traversal in the original GST, so that each node v receives the first time $\alpha(v)$ and the last time $\beta(v)$ visited during the traversal.[3] Then, each node u in the IGST is augmented with the information $\alpha(u')$ and $\beta(u')$ where u' is its correspondence in the GST. After that, for each IGST, we collect the set of α values of the nodes, and store a y-fast trie so that the successor query on the α values can be answered in $O(\log\log D)$ time.

Now, to perform searching, we first obtain the locus of P in the original GST, say u_P, whose pre-order traversal times are $\alpha(u_P)$ and $\beta(u_P)$. Since traversal times have a nice nested property,[4] to search for the locus of P in IGST-f, it is equivalent to finding the successor of $\alpha(u_P)$ in the set of α values of IGST-f. Precisely, let u be the node in IGST-f with $\alpha(u)$ being the successor of $\alpha(u_P)$. It is easy to check that $\beta(u) \leqslant \beta(u_P)$ if and only if the locus of P exists in IGST-f, with u being the locus. After the above successor query, we can check the corresponding 3-tuple (ℓ, r, c) of u to determine how many documents contain at least f occurrences of P. This gives the following theorem.

Theorem 3 *(Our proposed index). By storing the GST, and the I-arrays for all IGST-1, IGST-2, ..., and IGST-D, inverse_mine(P, k) query can be answered, for any input pattern P and input integer k, in $O(|P| + \log D \log\log D)$ time.*

As shown in [14], the number of nodes in IGST-f is $O(D/f)$ for any f. Briefly speaking, each leaf or each degree-1 node in IGST-f corresponds to a disjoint set of at least f suffixes of the documents in Δ, so that there are $O(D/f)$ of them. On the other hand, the number of remaining nodes cannot exceed the total number of leaves and degree-1 nodes, so that there are $O(D/f)$ of them. Thus, the total number of nodes is $O(D/f)$. This gives the following space complexity result:

Theorem 4. *The total space of the adapted index of Muthukrishnan (in Theorem 1), or our proposed index (in Theorem 3), are both $O(D \log D)$.*

Proof. The theorem follows since $\sum_{f=1}^{D} D/f = O(D \log D)$. □

Remarks. The above results clearly hold when the alphabet size $|\Sigma|$ is constant, as the traversal of P in the suffix tree can be done in $O(|P| \log|\Sigma|) = O(|P|)$ time. For general $|\Sigma|$, we shall augment a perfect hash table [4] in each internal node of the suffix tree so that each branch can be accessed in $O(1)$ time based on the branching character, and thus the traversal time remains $O(|P|)$. The hash tables require a total of $O(D)$ space so that the space complexity also remains unchanged.

4. Efficient index for top-*k* document retrieval problem

Once we have obtained an index for answering the *inverse_mine* query, we can find the set of documents in *top_document*(P, k), that is, those k documents with the most occurrences of P, by the following framework:

[3] We assume a global time which is incremented by 1 whenever a node is visited.

[4] The α and β values follow the property that for any u and v with $\alpha(u) < \alpha(v)$, either $\alpha(u) < \alpha(v) < \beta(v) < \beta(u)$ or $\alpha(u) < \beta(u) < \alpha(v) < \beta(v)$. This is also known as Parenthesis Theorem in [4].

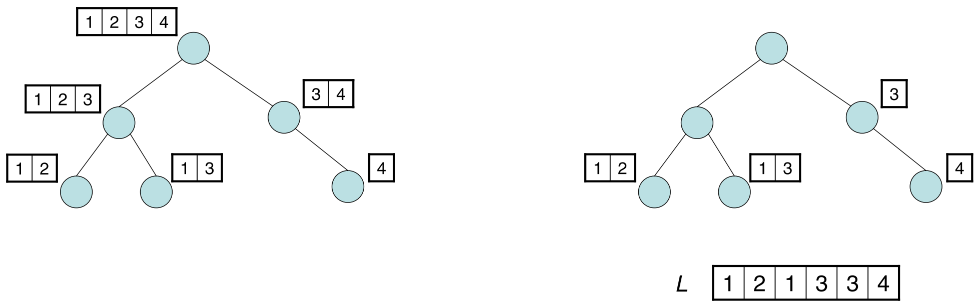

Fig. 3. Example of the concatenated list L. The two figures show the internal nodes and the sublist of each node in a hypothetical IGST-f. The left figure shows the original sublist associated to each node. The right figure shows the reduced sublist of each node after removing documents that appear in the node's proper descendants; the list L is formed by concatenating the reduced sublists along a pre-order traversal.

```
Solving $top\_document(P,k)$:

1. Find $f^*$ such that $f^* = inverse\_mine(P,k)$;
   /* Consequently, there are at least $k$ documents with at least $f^*$
   occurrences of $P$, but there are less than $k$ documents with at
   least $f^*+1$ occurrences of $P$ */
2. Output all documents with at least $f^*+1$ occurrences of $P$. Let $k'$ be the number of such
   documents;
3. Output $k-k'$ extra documents, distinct from those obtained in Step 2, which have at least
   $f^*$ occurrences of $P$.
```

One way to solve Step 2 is to augment each node v of IGST-f by the list of the associated $count(v)$ documents, each of which has at least f labels in the subtree rooted at v. Then, it is easy to see that in order to answer Step 2, we can just find the locus of P in IGST-(f^*+1), and output the list of documents in the locus. This method takes optimal $O(k')$ time in reporting the documents. Unfortunately, in the worst case, the extra space we need for the augmentation is $O(Dm\log D)$, where m denotes the number of documents in Δ. We refer this method as *Heuristic I*.

A better way to solve Step 2 is to apply Muthukrishnan's index for colored range query (Lemma 4 in Section 2) as an auxiliary data structure, as it is used in [14] for solving the document mining problem. The idea is that: For each node v in IGST-f, we only store the *reduced* sublist of the associated $count(v)$ documents, where each such document does not appear in the list of the proper descendant of v in IGST-f. Next, we perform a pre-order traversal, and concatenate the list of the visited node into a single list L. (See Fig. 3 for an example.) Then, it is easy to check that each node v will correspond to a subrange in the list L, such that its associated $count(v)$ documents will correspond *exactly* to the $count(v)$ distinct documents in the subrange. (For example, consider the left child of the root in the IGST-f in Fig. 3. It corresponds to the subrange $L[1..4]$, which is exactly the concatenation of the reduced sublists in all its descendants (including itself). We see that its associated documents-1, 2, 3-will correspond exactly to the distinct documents in $L[1..4]$.)

Thus, if we use Muthukrishnan's index for storing L, and for each node, we store the starting and ending positions in L for the associated subrange, Step 2 can also be answered optimally in $O(k')$ time, as in Heuristic I, but with a smaller $O(D\log D)$ space requirement.[5]

Step 3 can be solved similarly as in Step 2. We observe that any k documents with at least f occurrences of P, together with the k' documents obtained in Step 2, must contain a desired set of k documents for our $top\_document(P,k)$ query. So in Step 3, we will arbitrarily select a set of k documents with at least f occurrences of P, from which we further select $k-k'$ documents that are not obtained in Step 2. A simple way to solve the latter part is by sorting, taking $O(\min\{m, k\log k\})$ time. To speed up, we maintain an extra bit-vector of m bits, where the ith bit corresponds the document T_i, with all bits initialized to 0 at the beginning. When Step 2 is done, we mark each bit corresponding to the k' documents by 1. Then, in Step 3, when a document is examined in the latter procedure, we can check this bit-vector to see whether a document has been obtained in Step 2 already, so that we can easily obtain the desired set of extra $k-k'$ documents. After Step 3, we can simply reset the bit-vector in $O(k)$ time by referring to the final output. Thus, we have completely solved the *top_document* query, and have obtained the following theorem:

Theorem 5 *(Our proposed index). An $O(D\log D)$-space index for Δ can be maintained, such that for any input pattern P and input integer k, the query $top\_document(P,k)$ can be answered in $O(|P| + \log D \log\log D + k)$ time.*

[5] It is easy to check that the list L in IGST-f has $O(D/f)$ entries, since each entry corresponds a distinct f labels from the same document. Thus, the total space is $\sum_f D/f = O(D\log D)$.

Remark. Although Heuristic I does not guarantee good worst-case space bound, as our experiments showed, its space may be better than storing the color-range-query index in practice. We defer the details to Section 6.

5. Construction algorithms

In the above discussion, we have focussed on the design of the index and have not mentioned the construction time. In fact, the index of Theorem 1, which is equivalent to the index in [14] with *count* value augmented to each node, can be constructed by a simple adaptation of the construction algorithm in [14]. Once the count information in each node is available, the index of Theorem 2 can be constructed in $O(D\log D)$ extra time to traverse each IGST and build the corresponding array. Our index of Theorem 3 requires the construction of the y-fast trie; the bottleneck is to construct perfect hash tables [4] with a total of $O(D\log D)$ entries, and this can be done in randomized $O(D\log D)$ time. Thus, although the query time by the index of Theorem 2 is slightly slower than that of Theorem 3, the former index has a slight advantage (worst-case guarantee) in its construction time. In the following, we complete the details for the construction algorithm by showing how to augment the *count* values of the nodes in each IGST. This is done by directly applying Hui's algorithm [8] for the nodes in each IGST in a batch.

5.1. Augmenting the count values using Hui's algorithm

The construction algorithm in [14] allows us to obtain the topology of each IGST-f and the reduced sublist associated to each node, in $O(D\log^2 D)$ time. Based on this information, we can easily recover the original sublist for each node, using a bottom-up traversal in each IGST. More formally, suppose that our input collection Δ contains a total of m documents. Then in each node v, we can create a bit-vector $\mathcal{B}$ of size m such that $\mathcal{B}[i]=1$ if and only if the original sublist of v contains document i. The array $\mathcal{B}$ in each node v can be constructed by examining the reduced sublist of v, and the $\mathcal{B}$ arrays in all of its children. After that, the *count* value of v, which is the number of distinct documents in the subtree rooted at v, is exactly the number of 1's in the $\mathcal{B}$ array of v. In total, the above process to augment the *count* values in $O(D\log D)$ nodes in all IGSTs can be done in $O(Dm\log D)$ time. In fact, we can augment the *count* values more efficiently using an indirect method, first proposed by Hui [8]. The details are as follows.

Consider a particular IGST-f with the reduced sublist constructed. We now perform a preprocessing step:

1. Inside each node, write down the number of documents in its reduced sublist.
2. Perform a bottom-up traversal in the IGST-f, so that each node v obtains the total number n_v of documents in the reduced sublists of all its descendants (including itself).

The value n_v in each node v after Step 2 in the above preprocessing is closely related to the desired *count* value. While *count* value counts each distinct document in the descendants' reduced sublists once, the value n_v may count a document multiple times. To rectify the count, we have the following observation.

Observation 1. *(See [8].) Suppose that document i occurs in the reduced sublists of y nodes. Denote these y descendants by $u_1, u_2, \dots, u_y$ ordered according to their first appearance in the pre-order traversal. Suppose further that document i occurs in the reduced sublists of exactly j_i descendants of node v (including v itself). Then we have:*

1. *There exists an integer z such that the j_i descendants of v are equal to $u_{z+1}, u_{z+2}, \dots, u_{z+j_i}$.*
2. *The lowest common ancestor of u_g and u_{g+1} is in the subtree of v if and only if $g\in\{z+1, z+2, \dots, z+j_i-1\}$.*

The above observation immediately implies the following corollary.

Corollary 1. *(See [8].) Let j'_i denote the number of g's such that the lowest common ancestor of u_g and u_{g+1} is in the subtree of v. Then $j_i - j'_i = 1$ when $j_i \geqslant 1$, and $j_i - j'_i = 0$ when $j_i = 0$. In other words, $j_i - j'_i$ always indicates whether or not document i occurs in the subtree of v.*

Recall that n_v is the total number of documents in the reduced sublists in all descendants of v. Based on the above corollary, we can see that the *count* value for a node v-which is the number of distinct documents in the reduced lists in the subtree of v-can be calculated by

$$\sum_i (j_i - j'_i) = \sum_i j_i - \sum_i j'_i = n_v - \sum_i j'_i,$$

where i ranges over all documents.

The rectifying value of $\sum_i j'_i$ for each node v in the IGST-f can be found using a similar approach as we obtain n_v. The details are as follows:

1. Assign a counter to each node, which is initialized to 0.
2. Process each document i as follows.
   (a) Locate the nodes $u_1, u_2, \ldots, u_y$ which contain document i in its reduced sublist, where nodes are ordered by their first appearance in the pre-order traversal.
   (b) Add 1 to the counter in the lowest common ancestor of u_g and u_{g+1}, for all g.
3. Perform a bottom-up traversal in the IGST-f, so that each node v obtains the total number of times a descendant of v being a lowest common ancestor of some pair in Step 2.

The value stored in each node v after the above procedure is the desired value for $\sum_i j'_i$. Finally, by subtracting n_v by the corresponding $\sum_i j'_i$ in each node v, we obtain the desired *count* value for each node.

5.2. Time analysis

The preprocessing step to compute n_v can be done in time linear to the size of the IGST-f and the concatenated list L. The procedure to compute the rectifying value $\sum_i j'_i$ requires $O(|L|)$ lowest common ancestor queries, where each can be performed in $O(1)$ time if we build an LCA index [1] in advance. Such an LCA index can be constructed in time linear to the size of the IGST-f. Consequently, the procedure to compute the rectifying value can again be done in time linear to the size of the IGST-f and the concatenated list L. As mentioned before, the total size of all IGSTs and concatenated lists is $O(D \log D)$, so the total time to augment the *count* values is $O(D \log D)$. This gives the following result.

Theorem 6. *The adapted index of Muthukrishnan (in Theorem 1) can be constructed in $O(D \log^2 D)$ time, and our proposed index (in Theorem 3) can be constructed in randomized $O(D \log^2 D)$ time.*

6. Experimental results

In this section, we give the empirical tests of the practicality of the various indexes described in this paper. In the following, we first explain our experimental setup. Then we discuss the limitation of using our proposed index, and compare the performance of our index with the existing ones. Finally, we focus on our index and give a detailed case study for its performance.

6.1. The experimental setup

The Data Set: Our experiments were conducted with six types of text collections, whose details are as follows:

- `Random:`
  Each text in the collection is generated randomly, where characters are chosen from the English alphabet $[\mathtt{a}, \mathtt{z}]$ of size 26.
- `Zipfian:`
  We select 20 patterns, each of length 3, as words to be output by a random memory-less source, where the jth word has a probability of $(1/j)/\sum_{k=1}^{20}(1/k)$ to be output at a time. Each text in the collection is a sequence of words output by this random source.[6] The alphabet is $[\mathtt{a}, \mathtt{z}]$.
- `Fiction:`
  Each text in the collection corresponds to a distinct portion of the fiction "Harry Potter" (all 7 chapters). The alphabet is $[\mathtt{a}, \mathtt{z}] \cup \{\sqcup\}$ where $\sqcup$ denotes a white space.
- `Bible:`
  Each text in the collection corresponds to a distinct portion of the Bible. The alphabet is $[\mathtt{a}, \mathtt{z}] \cup \{\sqcup\}$ where $\sqcup$ denotes a white space.
- `Source Code:`
  Each text in the collection corresponds to a `.cpp` or a `.h` file in the source code of the peer-to-peer application *eMule* [6], downloaded from [17]. The alphabet is $[\mathtt{a}, \mathtt{z}] \cup \{\sqcup\}$, so that all special characters other than $[\mathtt{a}, \mathtt{z}]$ in a text are uniformly replaced by $\sqcup$.
- `DNA:`
  Each text in the collection corresponds to a portion of the Human mitochondria DNA, downloaded from [5]. The alphabet is $\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}$.
- `Web Page:`
  A total of 318 articles are selected from ESPN or other sport websites. Each article is related to the NBA games, such as players, teams, game previews, or reviews.

[6] The random source tries to simulate a generator for texts with words frequency distributed according to the Zipf's law [20], which is observed in most natural occurring texts.

Text Collection			Top-k Query		
source	# documents	# chars/document	pattern	k	*occ*
`Random`	100	4143	`aaa`	3	1
`Zipfian`	100	4143	most frequent pattern	3	40301

Fig. 4. The setting of the experiment on limitations.

Space (MB)			Query Time (microsecond)	
source	`GST`	`Ours(A)`	`GST`	`Ours(A)`
`Random`	16.5	82.9	250	125
`Zipfian`	18.2	106.0	12039	70

Fig. 5. The result of the experiment on limitations.

The Indexes: We implemented the various indexes described in the paper. They include the following:

- `GST`:
  The standard generalized suffix tree on the input collection.
- `Muthu`:
  The index proposed by Muthukrishnan [14] for document mining query. It can support top-k query in $O((|P|+k)\log D)$ time.
- `Adaptation`:
  The adaptation of `Muthu` (Index of Theorem 1) which contains *count* information. It can support top-k query in $O(|P|\log D+k)$ time.
- `Ours(A)`:
  Our proposed index with array representation for IGST-f (Index of Theorem 2). It can support top-k query in $O(|P|+\log^2 D+k)$ time.
- `Ours(YF)`:
  Our proposed index with y-fast trie representation for IGST-f (Index of Theorem 3). It can support top-k query in $O(|P|+\log D\log\log D+k)$ time.
- `Inverted Index`:
  We use two variants of the inverted index [22]. Both variants store a list of documents for each word, and for each document in the list, we store the positions in which the word appears in the document. The difference is in the order where we sort the documents. In the first variant, `Inverted Index (FS)`, documents in each list are sorted in terms of frequency of the corresponding word appearing in the document. This variant supports efficient *word query*, where the input pattern consists of a single word. In the second variant, `Inverted Index (DS)`, documents in each list are sorted in the document ids. This variant supports efficient *phrase query*, where the input pattern consists of a sequence of multiple words.

Platform: We implemented all the above indexes using the programming language `C++`, compiled with the `g++` compiler version 4.3.3. Our experiments were run on a `5200-AMD 2.7 GHz` machine with a 2 GB RAM. The OS was `Ubuntu` using kernel version `2.6.2.21`.

6.2. *Limitations*

The first experiment studies when we should use `GST`, and when we should use the various indexes described in the paper, for answering the *top_document* query. To compare with `GST`, we choose `Ours(A)` as a representative, though we expect similar results will be obtained when we use the other indexes. We test the behavior of the two indexes with `Random` and `Zipfian`. The experimental setting, and the result, are shown in Figs. 4 and 5 respectively:

The results show that, when the total occurrences of P, *occ*, in the text collection is very large, the query time is much better if `Ours(A)` is used than with *GST*. The discrepancy comes from the need of performing extra sorting the occurrences in `GST`, which takes $O(occ\log occ)$ time in theory. On the other hand, `Ours(A)` does not gain much in the other extreme, when the total occurrences of P is very small. In conclusion, `Ours(A)` should be used in those applications where we expect k to be much smaller than *occ*.

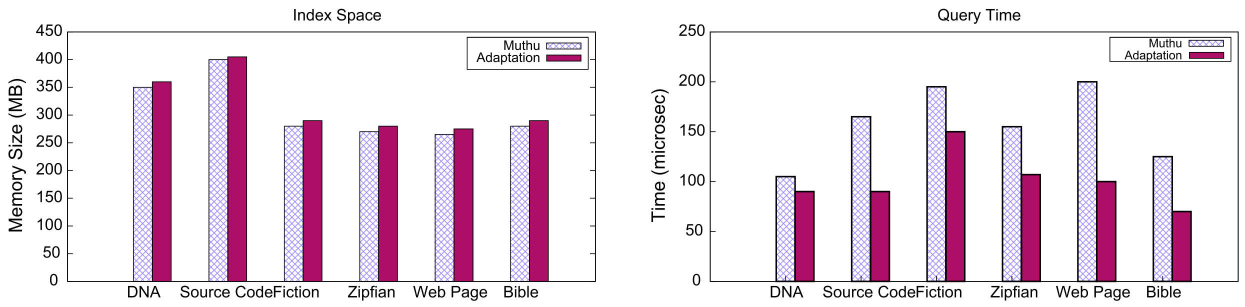

Fig. 6. Benefit of *count* information. The left chart shows the space for indexing with `Muthu` and `Adaptation`, respectively. The right chart shows the query time using these two indexes.

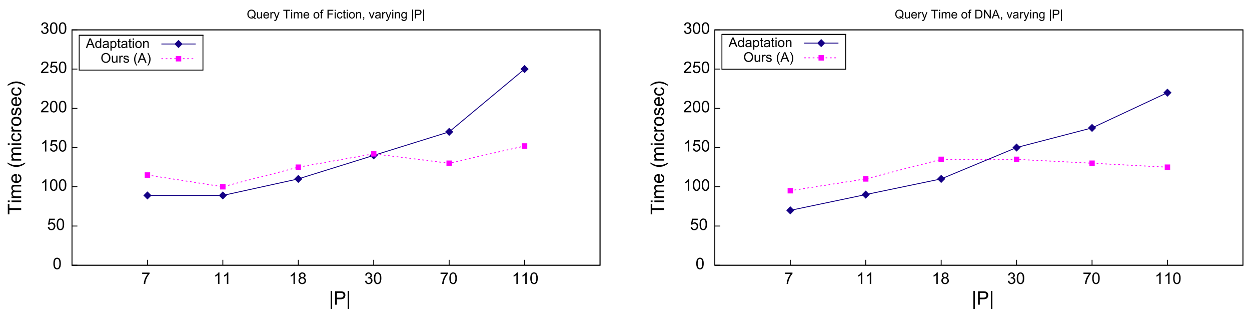

Fig. 7. Query time with different $|P|$. The left and right charts correspond to the query times when testing the indexes for `Fiction` and `DNA`, respectively.

6.3. *Performance comparison among various indexes*

6.3.1. *Benefit of count information*

The next experiment tested the benefit of storing the *count* information along with the IGSTs. We indexed various texts of similar size with the indexes `Muthu` and `Adaptation`, where the index-to-text ratio (in size) is roughly 250 times. The results of the experiment are shown in Fig. 6.

The results show that storing the *count* information only increases the total space by around 3%. On the other hand, the query time was improved significantly (from 20% up to 50%). This indicates that it is worthwhile to store *count* although it will slightly increase the storage space.

6.3.2. *Adaptation vs ours: Which one to choose*

The next experiment examined which of the indexes, `Adaptation` or `Ours(A)`, is better suited for top-k queries with different pattern lengths. We tested the two indexes for the texts `Fiction` and `DNA`, and set $k = 20$ for the top-k queries. The results are shown in Fig. 7.

The results show that `Muthu` performs slightly better when the pattern length is short, but as the pattern length increases, `Ours(A)` becomes better and then consistently outperforms `Muthu`. This agrees with the theoretical query time, where the query time for `Muthu` is $O(|P|\log D + k)$ while the query time for `Ours(A)` is $O(|P| + \log^2 D + k)$. The above results suggest that `Ours(A)` is generally a better choice unless we know in advance that the pattern lengths in most queries are short.

6.3.3. *Array vs Y-fast trie: Which one to choose*

The next experiment examined the performance of `Ours(A)` and `Ours(YF)`. We also included the performance of `Adaptation` as a control. We tested the three indexes for the texts `Fiction` and `DNA` with different size, and we set $|P| = 70$ and $k = 20$ for the top-k queries. The results are shown in Fig. 8.

The results show that despite the fact that `Ours(YF)` in theory has a better query performance than `Ours(A)`, which is $O(|P| + \log D \log\log D + k)$ versus $O(|P| + \log^2 D + k)$, `Ours(YF)` does not have much advantage for query time in practice. One possible explanation is that although y-fast trie has a good theoretical bound for predecessor/successor queries, its data structure is much more complicated than an array, making its practical performance bad. Also, `Ours(YF)` uses more space than `Ours(A)`. These results suggest that `Ours(A)` is generally a better choice than `Ours(YF)`.

6.3.4. *Ours vs inverted index*

We compared index `Ours(A)` with the two variants of `Inverted Index` in the next few experiments. We tested the indexes for the text `Bible`, and set $k = 5$ for the top-k queries. The word query consists of searching the highly frequent

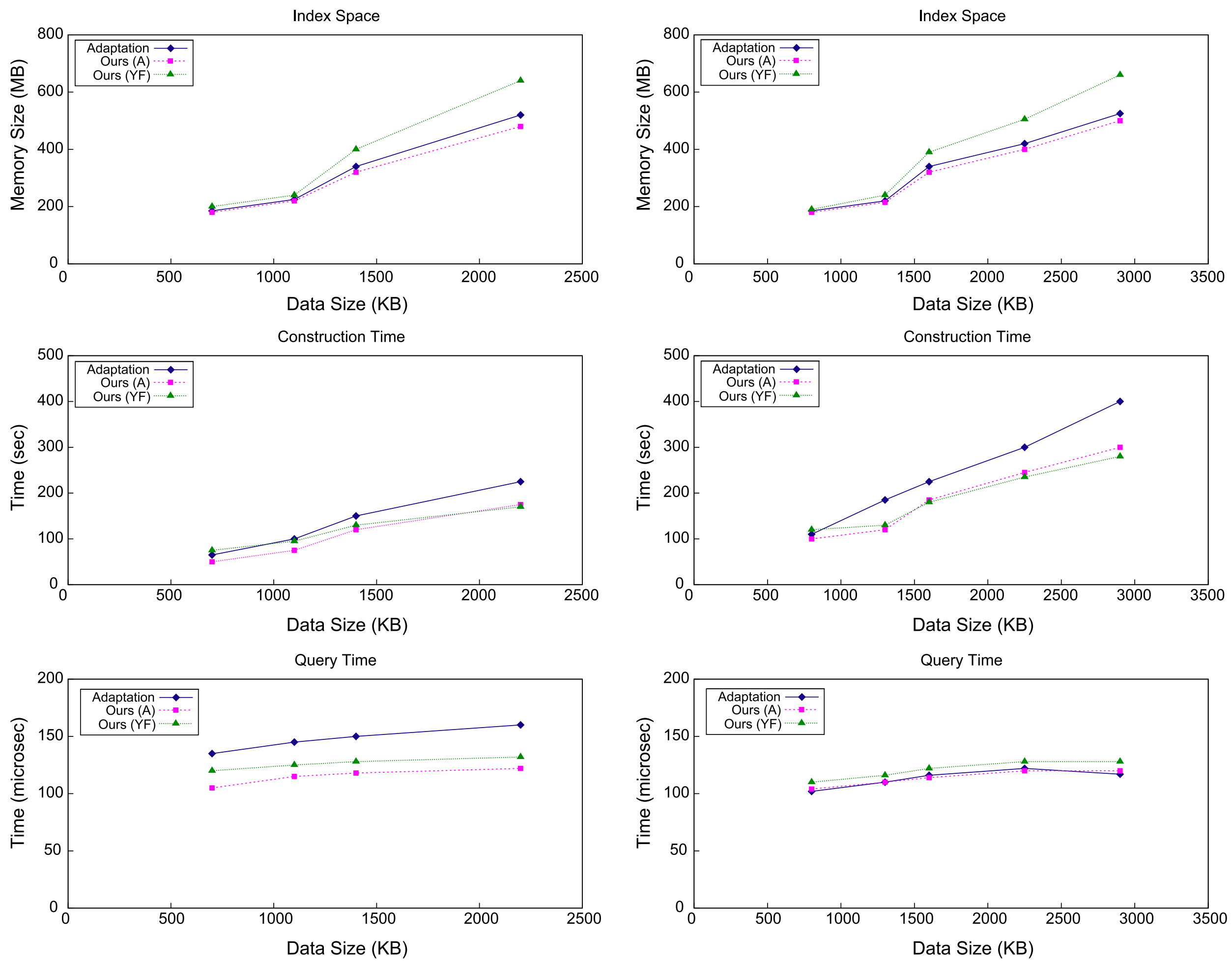

Fig. 8. Performance comparison (index space, construction time, query time) of `Adaptation`, `Ours(A)`, and `Ours(YF)`. The left and right charts correspond to the testing on these indexes for `Fiction` and `DNA`, respectively.

word (such as "the"), and the phrase query consists of searching of highly frequent phrase (such as "according to"). Our results are shown in Fig. 9.

The results show that `Ours(A)` achieves performance comparable to `Inverted Index` for word queries. However `Inverted Index` is not efficient for phrase queries as it needs to find the list of documents for each word in the query, and then computes the common documents that are found in these lists. Our results also reveal the sensitivity of `Inverted Index` towards number of documents in collection Δ (where we fix the average document size).

Better performance by `Ours(A)` against `Inverted Index` was at the expense of more space. In our experiments, we found that the size of `Ours(A)` is roughly 250 times of the input text whereas the size of `Inverted Index` is only 2 to 3 times of the text. The practical use of `Ours(A)` is limited by its heavy memory requirement. Nevertheless, we may replace the suffix trees in `Ours(A)` by the Compressed Suffix Trees (CST) [16] to decrease the memory usage.

6.4. *Case study: Our index with array representation*

The indexes described in this paper have a lot of similarities. Instead of examining each of them in detail, we choose `Ours(A)` as the representative for a more involved case study. We emphasize that similar results should be observed when we replace `Ours(A)` with the other indexes.

6.4.1. *Index space distribution*

The next experiment examined the index space distribution with respect to different components. We constructed `Ours(A)` for the texts `DNA` and `Fiction`. Fig. 10 shows the results.

From the results, we see that most of the space (around 40%) is consumed by IGST-1 and its corresponding CRQ index. Also, the space for CRQ indexes are much more than the space for IGSTs (the ratio is around 3 to 1). The latter implies that if only *inverse_mine* query is needed, 75% of the space can be saved.

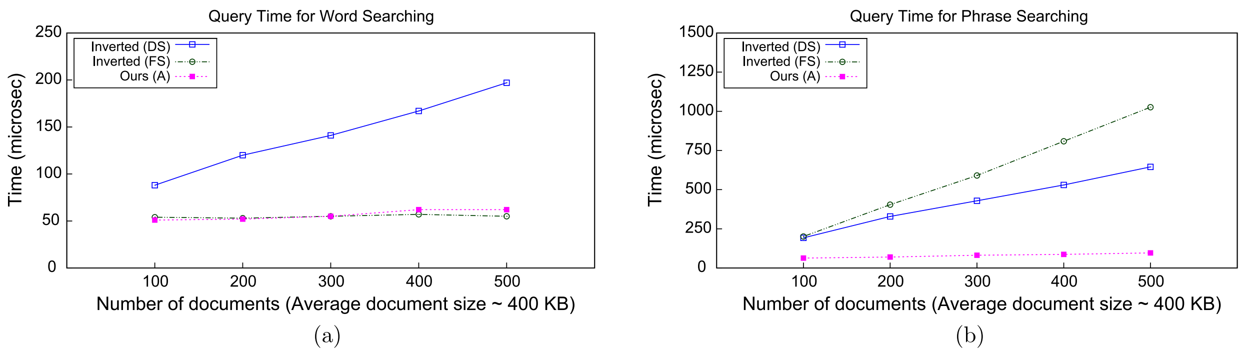

Fig. 9. Performance comparison of `Ours (A)` and `Inverted Index`. (a) Word query. (b) Phrase query.

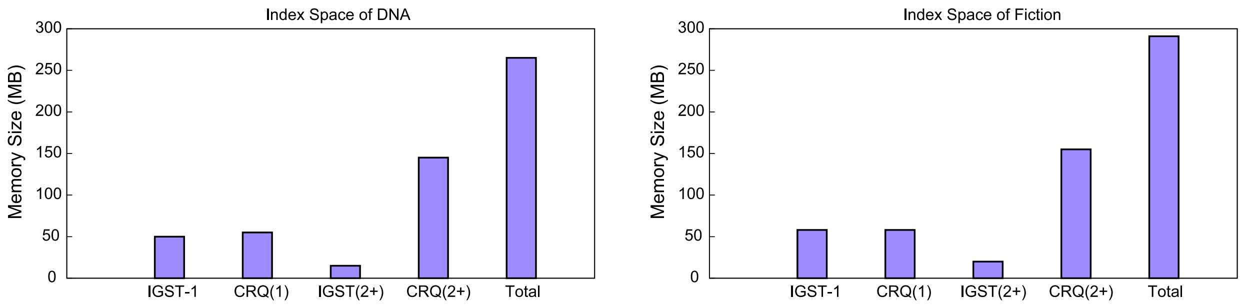

Fig. 10. Index space distribution with respect to components. The left and right charts correspond to the space for indexing `DNA` and `Fiction`, respectively. The entries x-axis in both charts refer to the components as follows. `IGST-1` := space for IGST-1, `CRQ(1)` := space for the CRQ index in IGST-1, `IGST(2+)` := total space for all IGSTs except IGST-1, `CRQ(2+)` := space for all CRQ indexes except the one in IGST-1, `Total` := total space of the index.

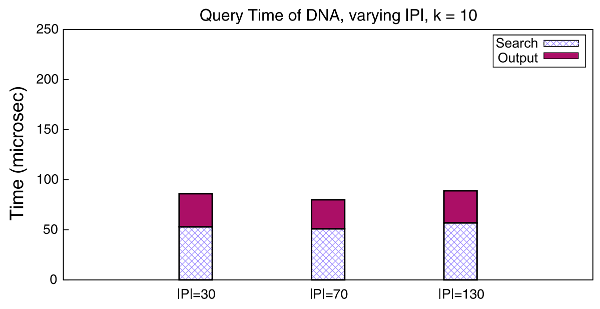

Fig. 11. Query time with different $|P|$.

6.4.2. *Query time distribution*

The next set of experiments examined the query time distribution with respect to searching and outputting the top-k documents. We tested the query times of `Ours(A)` for `DNA` on various pattern length, with $k = 10$. We also tested the query times of `Ours(A)` for `Fiction` on various k, with $|P| = 70$. Fig. 11 and Fig. 12 show the results.

From Fig. 11, we see that the query time grows very slowly as $|P|$ increases. One possible reason is that the time contributed by the $O(\log^2 D)$ term is far more than the time contributed by the $O(|P|)$ term. From Fig. 12, we see that the output time grows linearly as k increases as the theoretical bound suggests.

6.4.3. *Benefit of Heuristic I*

In Section 4, we mentioned an alternative approach, called Heuristic I, which avoids storing the CRQ index. Nevertheless, the index for Heuristic I does not have worst-case guarantee for its space bound. In the last experiment, we examined the practicality of such approach. We constructed an optimized version of `Ours(A)`, such that for each IGST-f in the index, we decide whether it takes less space to store the CRQ index or the index for Heuristic I, and store whichever is smaller. We compared this optimized version of `Ours(A)`, denoted by `Heuristic I`, against the original `Ours(A)` for the texts `DNA` and `Fiction`. Fig. 13 show the results.

From the results, we see that the optimized version of `Ours(A)` can reduce the space usage by around 40%. The query performance is *simultaneously* improved. This indicates that although Heuristic I may not have worst-case guarantee, it can be very useful in practice.

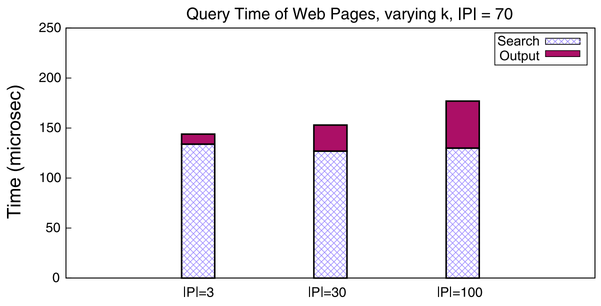

Fig. 12. Query time with different *k*.

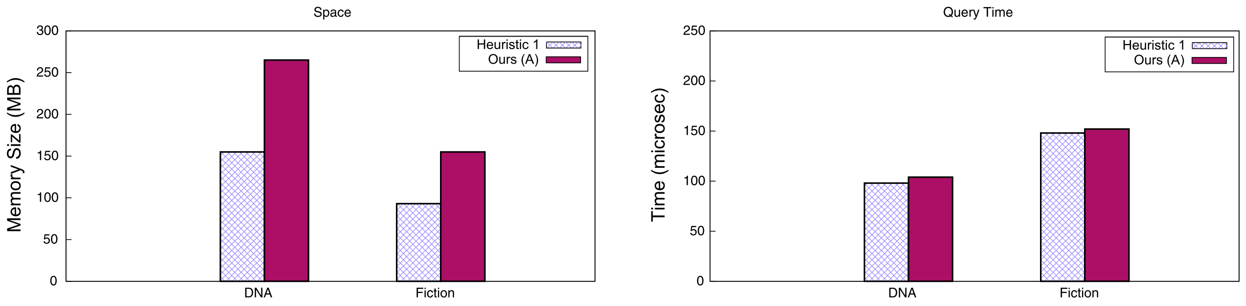

Fig. 13. Benefit of Heuristic I. The left chart shows the space for indexing with `Heuristic I` and `Ours(A)`, respectively. The right chart shows the query time using these two indexes.

7. Conclusion and open problems

We have introduced the *inverse document mining* and the *top-k document retrieval* problems, and proposed indexes which support the required queries in near-optimal time.

The core of our indexes, called the *induced generalized suffix trees*, are adapted from the ones by Muthukrishnan [14], where we store new sets of data to support our desired queries. In addition, we devise a simple array representation to organize the data, which consequently leads to an interesting way to answer queries. We have also conducted an experimental study for the practicality of our index and an adapted index of Muthukrishnan.

A main open question in the field is: Is there an $O(D)$-space index with nearly optimal query performance? And even better, can the index space be made close to the compressed form of the text?

References

[1] M.A. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of Latin American Symposium on Theoretical Informatics, 2000, pp. 88–94.
[2] I. Bialynicka-Birula, R. Grossi, Rank-sensitive data structures, in: Proceedings of International Symposium on String Processing and Information Retrieval, 2005, pp. 79–90.
[3] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (10) (1977) 762–772.
[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2001.
[5] Human Mitochondria Genome Database, Department of Genetics and Pathology, Uppsala University, Sweden, http://www.genpat.uu.se/mtDB.
[6] eMule, Wikipedia, the free encyclopedia (based on 28 July 2008 version), http://en.wikipedia.org/wiki/EMule.
[7] W.K. Hon, R. Shah, J.S. Vitter, Ordered pattern matching: Towards full-text retrieval, Technical Report TR-06-008, Department of CS, Purdue University, 2006.
[8] L.C.K. Hui, Color set size problem with applications to string matching, in: Proceedings of Symposium on Combinatorial Pattern Matching, 1992, pp. 230–243.
[9] R.M. Karp, M.O. Rabin, Efficient randomized pattern-matching algorithms, Technical Report TR-31-81, Aiken Computational Laboratory, Harvard University, 1981.
[10] D.E. Knuth, J.H. Morris, V.B. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (2) (1977) 323–350.
[11] V. Makinen, G. Navarro, Position-restricted substring searching, in: Proceedings of Latin American Symposium on Theoretical Informatics, 2006, pp. 703–714.
[12] Y. Matias, S. Muthukrishnan, S.C. Sahinalp, J. Ziv, Augmenting suffix trees, with applications, in: Proceedings of European Symposium on Algorithms, 1998, pp. 67–78.
[13] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM 23 (2) (1976) 262–272.
[14] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proceedings of Symposium on Discrete Algorithms, 2002, pp. 657–666.
[15] K. Sadakane, Succinct representations of *lcp* information and improvements in the compressed suffix arrays, in: Proceedings of Symposium on Discrete Algorithms, 2002, pp. 225–232.
[16] K. Sadakane, Compressed suffix trees with full functionality, in: Theory of Computing Systems, 2007, pp. 589–607.
[17] SourceForge, SourceForget.net: Open Source Software, http://sourceforge.net.
[18] N. Valimaki, V. Makinen, Space-efficient algorithms for document retrieval, in: Proceedings of Symposium on Combinatorial Pattern Matching, 2007, pp. 205–215.

[19] P. Weiner, Linear pattern matching algorithms, in: Proceedings of Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[20] E.W. Weisstein, Zipf's law, http://mathworld.wolfram.com/ZipfsLaw.html.
[21] D.E. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(N)$, Information Processing Letters 17 (2) (1983) 81–84.
[22] I. Witten, A. Moffat, T. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann Publishers, Los Altos, CA, USA, 1999.