2010

Implementation and analysis of a Top-K retrieval system for strings

Sabrina Chandrasekaran
Louisiana State University and Agricultural and Mechanical College

IMPLEMENTATION AND ANALYSIS OF A TOP-K RETRIEVAL SYSTEM FOR STRINGS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science
The Interdepartmental Program

in

The Department of Computer Science

by
Sabrina Chandrasekaran
B.Tech., National Institute of Technology, Surathkal, India, 2008
August 2010

Acknowledgements

My sincere thanks to my advisor Dr. Rahul Shah for his patience and guidance throughout the course of my research. His methods of intuitive thinking and research methodology have provided a great source of inspiration to me. I would also like to thank Dr. Rajgopal Kannan and Dr. Konstantin Busch for being a part of my committee and offering their guidance and support.

I would like to thank Manish Patil and Sharma V.T. for their help and guidance in the implementation. This work would not have been possible without the discussions and help that they provided.

I am also grateful to Dr. Yejun Wu of the School of Library and Information Science for us with data sets to test our index on. I am also thankful to Simon Gog, a doctoral student with the University of Ulm for providing us with components of code to perform Compressed Suffix Tree and bit vector dictionaries. I am also grateful to him for his patience and helpfulness while guiding me on the use of his implementation. I am grateful to K.Sadakane, Associate Professor at the National Institute of Informatics, Japan for providing us with his implementation of bit vector dictionaries.

I wish to also thank my parents for their constant love, support and encouragement.

Table of Contents

Abstract

Given text which is a union of d documents of strings, $D = d_1, d_2, ..., d_d$, the emphasis of this thesis is to provide a practical framework to retrieve the K most relevant documents for a given pattern P, which comes as a query. This cannot be done directly, as going through every occurrence of the query pattern may prove to be expensive if the number of documents that the pattern occurs in is much more than the number of documents (K) that we require. Some advanced query functionality will be required, as compared to listing the documents that the pattern occurs in, because a defined notion of "most relevant" must be provided. Therefore, an index needs to be built before hand on T so that the documents can be retrieved very quickly. Traditionally, inverted indexes have proven to be effective in retrieving the Top-K documents. However, inverted indexes have certain disadvantages, which can be overcome by using other data structures like suffix trees and suffix arrays.

A framework was originally provided by Muthukrishnan [29] that takes advantage of the number of relevant documents being less than the occurence of the query pattern. He considered two metrics for relevance:frequency and proximity and provided a framework that took $O(n \log n)$ space. Recently, Hon et al [14] provided a framework that takes $O(n)$ space to retrieve the Top-K documents with more optimal query times, $O(P + K \log K)$ for arbitrary score functions. In this thesis we study the practicality of this index and provide added functionalities, based on the index, to retrieve Top-K documents for specific cases like phrase searching. We also provide functionality to output the K most relevant documents(according to page rank) when two patterns are given as queries.

Chapter 1

Introduction

Since its inception, the Internet has grown far and wide. The expanse of information available at our finger tips is mind boggling. In fact it has been stated that if one wished to read the Internet it would take him fifty seven thousand years (given he never stops reading). This gives us an estimate of how much information is actually there. No matter what we want to search for, we just need to type in a few relevant words into any search engine and we get millions and millions web sites which contain the information that we need. As an example, a search for the word 'thesis' on google.com gives 55,200,000 results in 0.19 seconds. The Internet search engines manage to find pages that contain the information that we need and return them in some relevance order. What is even more amazing is that it does all this under a second. Searching through millions of documents and ranking these results (such that the most important are returned first) most definitely cannot be done directly when the query is given. In order that it does all the above and does it fast we may need some specialized indexes. This is our focus in this thesis.

Information retrieval (IR) is termed as retrieving meaningful or related information from a huge pool of information or data. This text can be a union of a number of documents and thus our problem can also refer to extracting documents that contain the meaningful information we are looking for. This is termed as Document retrieval. This leads us to our fundamental problem: Given a text of size N and a pattern of size P, find all locations in text where this pattern occurs. Formally we say that the text, T is a union of D documents, where $d = d_1, d_2, d_3, ..., d_D$ is a collection of documents of total size N. The strings in these documents are drawn from an alphabet set Σ. When the text is given before hand and the queries come online (dynamically) it might be help to build a data structure on the text, such that when the queries come in, retrieval is fairly fast $O(P + occ)$ time [9].

A retrieval system must thus:

1. Find the documents that contain the query (pattern match).

2. Create an index that can return these documents according to some relevance ranking.

The most commonly used index in web search engines is the inverted index data structure. Inverted indexes require you to define the terms used in text (which would be words in simple English texts) [16]. Each of these terms or words is associated with a list of documents from the document set, d. When the query is given, only the documents associated with the query

need to be printed. These documents can also be stored in some relevance order if required. Seems simple enough, however, this leads to one major problem, the query must compose only of terms (as defined) in the index i.e.this technique will only work if documents consist of distinct words. While for most natural language documents this might work, it is difficult to make it work for a more general case where documents consist of a sequence of symbols without any word boundaries. Some examples of these situations are when the documents consist of a Far eastern language, which may not be easily parsed into words, or biological data, like protein or DNA data. Natural language text derived from OCR or speech-to-text systems may not also contain easily definable terms. OCR writes it as it sees it. A common source of problems is the hyphenations. OCR, being unable to read it, splits the words into two words. This can increase the index size and the query word can only be retrieved if it is an inverted index was built for phrase searching. Stopwords are not normally indexed by inverted indexes, but stopwords can be read wrong and end up getting indexed. This can be a further waste in space. Overall, inverted indexes may prove to be infeasible in certain cases (as mentioned above) and puts restrictions on queries.

For biological applications, n-gram analysis is required on the text and a specialized n-gram based inverted index may have to be used for the same. There are two ways in which we may be able to extend inverted indexes to be able to perform either substring searches or n-gram analysis:

1. Create an inverted index for all substrings of D
2. Create an inverted index over all possible n-grams of D, i.e. over all sub-strings of D of length n.

The space requirement for approach (1) might prove to be quadratic, thus making it inefficient. As far as approach (2) goes, the searches are now restricted to queries of size n. If pattern P turns out to be of length $m > n$ the retrieval is performed by analyzing every n gram in the list [25]. Each n- gram must also be associated with a list of positions where it occurs in text T. The documents, that the pattern P occurs in, can then be determined by merging all occurrence lists of all the n-grams, which P consists of. This can prove to be very time consuming, as the longest list can be much longer than the number of occurrences of pattern P.

If one can afford to use a little more space, we can negate all of these problems by using other full-text-indexes like Suffix Trees [31] or its more space efficient alternative, Suffix Arrays [24].

The Suffix tree is a data structure that presents all the suffixes of a given text (or string) in the form of a tree for particularly fast identification. The tree is stored such that any suffix, S from the text will be represented by exactly one path from the root to a certain node. Strings are represented by edges on this tree. Identification or retrieval time is therefore directly proportional to the size of the pattern (P) you are searching for. Suffix Arrays are a more space efficient representation of the Suffix Tree. They store the starting positions of all suffixes present in text,in lexicographic order, in an array of integers. Further explanations of suffix trees and suffix arrays are provided in Chapter 2.

When it comes to search engines, most queries are *bag-of-word* queries. You are more likely to refine your search by searching for 'thesis suffix tree' rather than 'thesis' and then

search for the one's with suffix tree. Some queries are proper nouns, and cannot be handled by a *bag-of-word* model. Some queries have phrases marked in quotes [16]. For all these kind of queries it might be more advantageous to use a suffix tree.

We now need to explain what we mean by 'relevant documents' in information retrieval. Relevance Ranking refers to the ranking of the documents in some order, so that the result returned first is what the user wants to look at most. This can be the document where the given query occurs most number of times (frequency). It can also be defined by a similarity metric, which can be the proximity of one pattern to a certain word or another pattern. Another form of ranking is by `TF-IDF` score; `TF` stands for term frequency (number of times the word occurs in document,d) and `IDF` stands for inverse document frequency (logarithm of the inverse of document frequency for query). The `TF-IDF` score is the product of `TF` and `IDF`. If a certain term appears in lower number of documents its idf score will be high and therefore the product will be high and if it is a term occuring in higher documents its idf score is decreased. It then stands that the Top-K documents are returned ordered by the similarity metric which ensures that a high frequency in a certain document means it has a higher ranking and a high frequency over all documents reduces its contribution. Another ranking technique is the page ranking scheme, where every document has a certain weight(in the case of web searching it could be how many times this document was found relevant or was retrieved for this particular query). Thus, the Top-K documents would then be the documents with the highest weight or pagerank value. Further explanations of different indexing techniques are provided in chapter 2.

This thesis is organized as follows: Chapter 2 explains the Basic Concepts and Preliminaries including various indexing methods available. It also goes into other work related to this Thesis. Chapter 3 explains the Index Construction and the different components needed to construct it. Chapter 4 describes the implementation carried out along with different compression techniques and how we applied them to our index. Chapter 5 discusses the experimental analyses that we carried out on the index. Finally, Chapter 6 summarizes and provides a conclusion along with outlining further work that can be carried out.

Chapter 2

Preliminaries and Basic Concepts

2.1 Suffix Trees

The best known indexing technique for data that cannot be broken down into words is the suffix tree[22]. On traversing the entire string, we define a suffix as the string we obtain from each position to the end of the text. Thus for a text of size N we have N suffixes. As an example we take the string = BANANA. The string used in a suffix tree is normally terminated by a '$', which is smaller in value than any other character in the alphabet set Σ. The string thus becomes : BANANA$. The suffixes for this word, arranged in lexicographic order, are:

1. A$
2. ANA$
3. ANANA$
4. BANANA$
5. NA$
6. NANA$

As can be seen from the figure 2.1 the suffixes are all represented in the suffix tree in the same lexicographic order as listed. If these edges are hashed (for example according to the node they originate from and the starting letter of the edge) they can be quickly identified in $O(1)$ time. We can, thus, identify the entire pattern in $O(P)$ time.

Suffix Tree requires the storage of a tree along with the edges and the nodes. Instead of storing one character per edge, as would be done in a trie, space is conserved by using a technique called path compression[31]. Paths that have only one child are compressed by removing the nodes in the middle. Along the edges the strings are stored and, to conserve space, position of the starting character and the length of the string are stored on the edge. These edges are mapped to the node that the edge originated from and the node it led to. At each node we need to keep track of its children, its sibling node and its parent node.

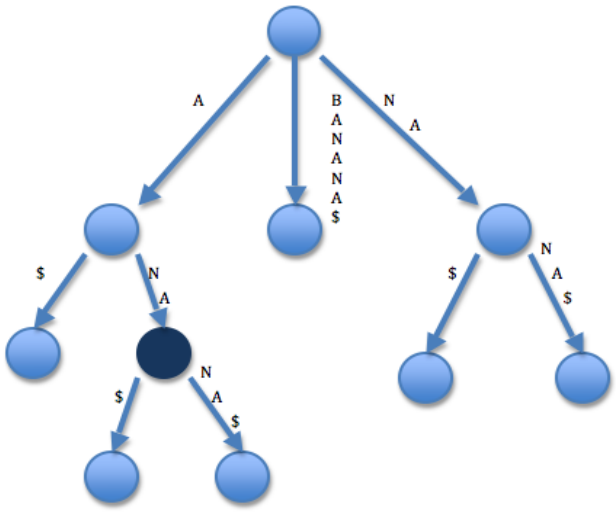


Figure 2.1: A Suffix Tree

As can be seen in 2.1, the pattern P can be matched easily. For example, if we take a pattern $P = AN$. The pattern is matched by first searching through the root node's children for an edge that starts with the letter A. Once such an edge is identified, the end node of this edge is identified (end node is node 2 in the example). End node's children are then traversed for an edge which starts with the letter N. As can be seen in the figure the pattern P is said to match at node 4, the locus of the pattern. Because of the lexicographic sorting,suffixes that start with a given substring (which is our pattern) will form a range $A[sp, ..., ep]$, where A refers to an array of all the leaf nodes and sp and ep refers to the leaf numbers of the range where the pattern matches ($sp <= ep$). The number of occurrences of the pattern can then be calculated by occ= ep-sp+1. For this suffix tree sp=2 and ep=3(where $0 < sp : ep < n-1$) and occ= 3-2+1=2.

To find each suffix, of course, we will have to traverse the entire string atleast once. If we take a position i, $0 < i < n - 1$, the tree can be updated with the new suffix which is the string from position i to n. On updating the tree with the new character, that came in, we also need to update all positions where suffixes of the suffix being updated exist with this new character. If we traverse each edge in the tree,searching for suffixes, the algorithm would be an $O(n^2)$ algorithm. To avoid traversing unnecessary edges suffix links are created. By definition, a suffix link is a pointer to the edge which contains the same string except for the first character[31]. The end node for the edge representing banana$ has an arrow going to an edge which represents anana$. This is the suffix link. The suffix link for the edge anana$ in turn would contain a pointer to edge nana$ and so on. If you then want to update all the edges which are suffixes of the edge being updated just follow the suffix link till you hit the root node (at which point the root node may have to be updated if the character is not already present as its child). Thus, the construction algorithm is a linear time $O(n)$ algorithm. However,this ease in identification comes with an expense. The space required to store this suffix tree, though proportional to the size of text, can be up to 50 times the text size (practically).

2.2 Generalized Suffix Trees

The above definition of a suffix tree can index text from a single document. For our case, we consider text which is a union of documents i.e. we have a collection $D = d_1, d_2, ..., d_d$ of d strings, of total length N. Thus the suffix tree needs to be built on the entire collection of d documents. Thus the documents are combined into one long document, with a unique differentiating character (something that doesn't occur in the rest of the text) between documents. This is then indexed into a suffix tree and an external mapping system is used to filter the final results. Each position of text can be mapped to the correct document id.

2.3 Suffix Arrays

A more space efficient version of the suffix tree is the Suffix Array [24]. It follows relatively the same principal as the Suffix tree only that it doesn't store the information in the form of a tree. It stores it as a table as shown in 2.1 for a text T= abracadabra$

Table 2.1: Suffix Array - An Example

No.	Index	Sorted Suffix	lcp
1	12	$	0
2	11	a$	0
3	8	abra$	1
4	1	abracadabra$	4
5	4	acadabra$	1
6	6	adabra$	1
7	9	bra$e	0
8	2	bracadabra$	3
9	5	cadabra$	0
10	7	dabra$	0
11	10	ra$	0
12	3	racadabra$	2

The suffix array is [12 11 8 1 4 6 9 2 5 7 10 3].

$SA[i]$ stores the starting position of the i^{th} lexicographically smallest suffix in T. The space taken by the suffix array is only about four times text. Saving on space comes with a price. A binary search needs to be performed over the array to identify the presence of the pattern. Matching the given pattern using the above array will take $O(P \log n)$ time as compared to $O(P)$ time that the suffix tree takes. If the text is a union of documents, you may need an external mapping system to find the documents the pattern occurs in.

In Table 2.1 [2], we see an entry for lcp, Longest Common Prefix. The LCP array stores the longest common prefix, the number of letters in the prefix common between $SA[i]$ and $SA[i-1]$. For example, the longest common prefix between $SA[3]$ and $SA[2]$ is $abra$ of size 4. By using this information, pattern matching becomes a lot easier, as comparing characters

that you already know exist need not be searched through when searching through the suffixes. Using this information pattern matching can be done in $O(P + \log n)$.

2.4 Word Suffix Trees

The advantage of using suffix trees/arrays is that every suffix does not need to be indexed. Indexing can also be done at certain points such as the beginning of each word. This leads to a specialized suffix tree called the word suffix tree [3] and can lead to a considerable decrease in the space taken by the suffix tree. For our purpose, we use the word suffix tree for a specialized index to retrieve Top-K documents for phrases. Figure 2.2 shows an example of a word suffix tree. So, if the query will only consist of whole words (in the case of phrase searches) we don't unnecessarily index other suffixes that won't be required for query.

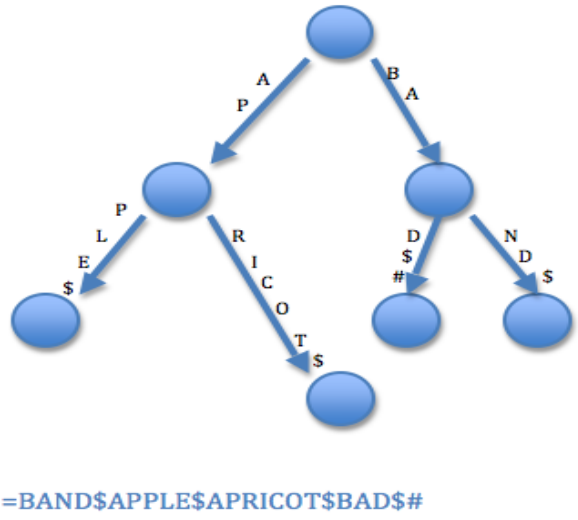


Figure 2.2: Word Suffix Tree

The space saving is apparent in the figure 2.2. The word separator used in this figure is a dollar ($). Thus only suffixes that start with a character immediately following the word separator, $ in this case, are updates in the suffix tree.

The above given methods work perfectly and provide full text indexes so that we can identify the documents where the pattern occurs more efficiently. However, practically, space does provide a concern and this has lead to the creation of compressed indexes where the data structures are stored in compressed form. From the creation of Suffix Arrays more compressed text indexes like Compressed Suffix Trees (CST) and Compressed Suffix Arrays (CSA) were created.

2.5 Compressed Data Structures

A data structure that uses considerably lesser space (close to entropy compressed space) and still manages to perform operations almost as fast as the uncompressed data structures are

called compressed data structures. Examples of compressed data strutures are Compressed Suffix Arrays [27]/ Trees [19], FM Index[11] etc. All these indexes can find the number of occurrences of pattern P by finding the sp and ep values of the pattern (explained in section 2.3).

2.5.1 Compressed Suffix Array

The Compressed Suffix Array (CSA) provides a compressed form of the suffix array. Suffix Array's take $n \log n$ bits and this space is reduced to $O(n \log |\Sigma|)$bits. Access time, however, increases from constant time to $(O(\log^{\epsilon} n))$. This was originally proposed by Grossi and Vitter [27]. Sadakane [17] modified it to an index who's space is proportional to the zeroth order entropy and self indexing(can reproduce the text if needed). Basic operations performed by the CSA are:

COMPRESS(T, SA): The Suffix Array is created using the text T. The suffix array is then compressed and saved while the original uncompressed suffix array can be discarded. The text, however, must be saved.

LOOKUP(i): This gives the value of $SA[i]$ from the compressed index.

The build time is $O(n)$ and the look up time is $O(\log \log n)$.

2.5.2 Compressed Suffix Tree

Compressed Suffix Tree (CST) [19] provides all functionalities of the original suffix tree but only with much lesser space (proportional to entropy compression of the data). It is thus a very good substitute for the suffix tree as it reduces space and provides all functionalities of the tree including the ability to perform substring matches and indexing text that cannot be disintegrated into terms. It thus provides a very good edge over the inverted index. It also has a self indexing property which enables us to discard the original text (in comparison to the inverted index where the original text must be kept). These indexes can return the values of sp and ep using which we can determine the occurrence of pattern P. Any operation that a suffix tree can perform can be performed by the CST with a slowdown of a factor of $polylog(n)$.

The Compressed Suffix Tree is essentially compossed of three components:

Compressed Suffix Array(CSA): As explained above a Compressed Suffix Array is a compressed version of the suffix array. Based on CSA, Compressed Suffix Tree's were developed. The functions required for the CSA to be applied to a CST are :

1. lookup(i) :Explained in section 2.5.1.
2. inverse(i): Returns $j = SA^{-1}[i]$ such that $SA[j] = i$.
3. $\Psi[i]$: $SA^{-1}[SA[i] + 1]$
4. substring(i,l): Returns $T[SA[i]..SA[i] + l - 1]$

The function $\Psi[i]$ in the compressed suffix array is defined as:

$$\Psi[i] = \begin{cases} i' & \text{if } SA[i] < n \text{ (such that } SA[i'] = SA[i] + 1 \text{)} \\ 0 & \text{if } SA[i] = n \end{cases}$$

Space of the CST in bits is $O(nH_0 + n \log\log \Sigma)$. Query time to retrieve the SA value, t_{SA} is $O(\log^{\epsilon} n)$ and t_{Ψ} is $O(1)$[17].

Balanced Parantheses Representation of Trees: A rooted tree which consists of n nodes can be encoded in $2n + o(n)$ bits. Such an encoding enables navigational operations also to be performed. Thus a suffix tree for a text of length n has $2n-1$ nodes and can be encoded in at most $4n+o(n)$ bits [23]. Further explanation on Balanced Paranthesis encoding is provided in section 4.1.1.

LCP Array and Height Array: It has been shown [30] that the traversal of a suffix tree can be simulated by using the suffix array and the array storing a set of the lengths of the longest common prefixes between two suffixes,called the Hgt array.

$$Hgt[i] = \begin{cases} lcp(T_{SA[i]}, T_{SA[i+1]}) & 1 \leq i \leq n-1 \\ 0 & \text{i=n} \end{cases}$$

Hgt requires $n \log n$ bits to store these values. Even though the values are small the largest value stored can be $n - 1$. In [17] it was shown that using a data structure of size $2n + o(n)$ bits a value $Hgt[i]$ can be computed in constant time, thus providing an space-efficient data structure for storing lcp values.

2.5.3 FM Index

FM Index is a Compressed index which provides pattern matching functionality. This index is based on Burrows Wheelers Transform (an algorithm used in compression techniques like bzip2). This index occupies space proportional to the entropy of the input data and does so with no significant slowdown in query time. It performs substring searches (unlike the inverted index) and is self indexing (can reproduce parts of the text by only decompressing portions of it) thus leading to significant space savings. The index originally occupied $5nH_k(T) + o(n)$ bits of storage, where H_k is the K^{th} order of entropy, and performs query in $O(p + occ\log^{1+\epsilon} n)$, where occ is the number of times the pattern occurs in text and $\epsilon > 0$. It can find the number of occurrences of the pattern and locate this pattern ($O(\log^{c} n)$), extract a part of the text and can display L characters surrounding the pattern P [11]. A variant of FM Index was later introduced [10] which took $nH_k(T) + o(n)$ bits of storage. This holds for all $k <= \alpha \log_{|\Sigma|} n$, constant $0 < \alpha < 1$ and $|\Sigma| = O(polylog(n))$. Occurence of a pattern P can be found in $O(P)$ time and it locates each pattern in $O(\log^{1+\epsilon} n)$ time for any constant $0 < \epsilon < 1$ and reports a text substring of length l in $O(l + \log^{1+\epsilon} n)$ time.

As an example, if $T = mississippi\$$ and we want to find pattern $Q = ssi$, the Burrows Wheeler's Transform is shown in 2.3 [8] :

The Last column, L, is $BWT(T) = ipssm\$pissii$. Using this information, the text is trasformed into another string, $BWT(T)$, which is easier to compress. From this string, the

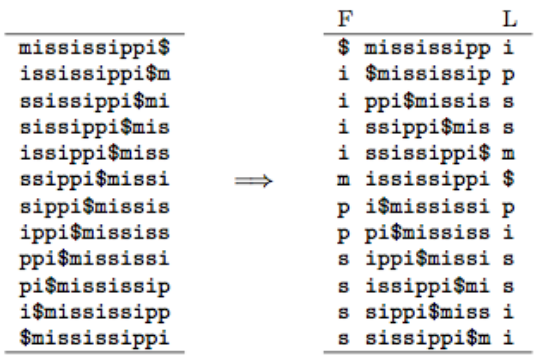


Figure 2.3: BWT for Text=mississippi$

text can be retrieved using a mapping function between F and L [10].
$LF(i) = C[L[i]] + rank_{L[i]}(L, i)$
where $C[c]$ counts the number of symbols less than c in the string L and $rank_c(L, i)$ counts the occurence of c in the string $L(1, i)$. Array L occupies $O(|\Sigma| \log n)$ space.

Finding the sp and ep values or the suffix range where the pattern occurs is called "Backward Search". The pattern is read backwards i.e. the pattern is read in the order iss. First the suffix range for i is found and this is transformed to the suffis range where si is found and so on. Assume that you have the suffix range for i, sp=2 and ep=5. Within this range we look for entries where L has the value s, as these are the entires where, in the original text, you get substring si. Using the formula $sp = C[c] + rank_c(L, sp - 1) + 1$
$ep = C[c] + rank_c(L, ep)$

We can intuitevely see that C[c] would take you to the position where s starts (as C[c] returns the number of symbols less than s) and $rank_c(L, sp - 1) + 1$ would take you to the position where, in L, i starts within the suffix range for s thus giving you the beginning of the suffix range for si. ep can also be mapped in the same way. In this way if you go through the entire length of the pattern you can find the required suffix range, sp and ep for the pattern.

A related notion to compressed data structures is the succinct data structure. This refers to a data structure that uses space close to the information theatrical lower bound while the space of the compressed data structures is proportional to the size of data given.

2.6 Inverted Index

The Inverted Index is one of the most commonly used indexes to retrieve documents containing the specified pattern. The input text is disintegrated into terms and for each word a list that contains all the documents it occurs in is stored. Thus to find the documents that the query occurs in we just need to output its document list [16].

The words are stored in alphabetical order, so that finding the word in the list is faster. Document id's are stored in ascending order so as to make use of this property to compress the list. When a list is stored in ascending order, gap encoding can be used explained in

Algorithm 1 SEARCH PERFORMED USING BWT-FM INDEX [12]

Backwards_search(Q[1,q])
$i = q$, $c = Q[q]$, $sp = C[c] + 1$, $ep = C[c + 1]$;
while $sp \leq ep \quad and \quad i \geq 2$ **do**
  $c = Q[i - 1]$;
  $sp = C[c] + rank_c(L, sp - 1) + 1$;
  $ep = C[c] + rank_c(L, ep)$;
  $i = i - 1$;
end while
if $ep < sp$ **then**
  return "Pattern not Found";
else
  return $[sp, ep]$
end if

section 4.2.

There are several variations when it comes to inverted index, different implementations according to what we may require off of the index. If you want to retrieve the Top-K documents you could store, along with the doc id's, the frequency of that word in each document (Frequency Ordered Lists). This list can be stored in decreasing order of frequency for quick retrieval of the Top-K frequencies. However, it is important to point that since it is no longer stored in increasing order of document id's, it becomes harder to compress these lists.

Another variation of the Inverted Index is the storage of a position list to be able to perform phrase queries and proximity searches(Position Ordered Lists). Position's can be at any preferred granularity either on a word level or paragraph level (according to the proximity that may be required in the query). It is more common to store the position's at a word level. It is harder to compress these lists as position lists will be more haphazard. As an example let us assume[1]:
$T0$ = "it is what it is", $T1$ = "what is it" and $T2$ = "it is a banana". The inverted index will be as follows (0,1 and 2 refer to the document id)

Frequency Ordered Listing
"a": 2
"banana": 2
"is": 0, 1, 2
"it": 0, 1, 2
"what": 0, 1

Position Listing
"a": (2, 2)
"banana": (2, 3)
"is": (0, 1), (0, 4), (1, 1), (2, 1)
"it": (0, 0), (0, 3), (1, 2), (2, 0)
"what": (0, 2), (1, 0)

Thus, if P= "what it is the output would be:
first list (without positions): 0,1
second list (with positions): 0

The space taken by an inverted index can vary form 5% of the text to 100% due to the the various implementations and variations available. But as inverted indexes are word based they are not suitable for data with no word boundaries. And, as is seen above, they aren't very efficient for phrase searches as they need to find the list of documents for each word in the phrase and then apply the position list to find the documents where the words occur consecutively in the correct order (as shown in example 2 with the word list).

2.7 RMQ Index

In our index we use a Range Maximum Query (RMQ) index to find the documents where our query pattern occurs most frequently. Definition : Given an array, A of numbers. A can be preprocessed such that to create a linear space data structure that can give you the position and value of the maximum value in a sub range $A[i, .., j]$ such that $0 <= i, j <= N$, where N is the number of numbers in array A. We use this property to find the largest K numbers given t such ranges.

RMQ schemes normally use Cartesian tree and LCA to obtain the minimum within a range. All schemes give the answer in $O(1)$ time. Berkman and Vishkin proposed a data structure which takes $O(n \log n) + |A|$ bits, where $|A|$ is size of the input array A[26]. In [28], Schieber and Vishkin introduced a data structure that also took $O(n \log n) + |A|$ space. A simpler scheme was later shown in [6] by Bender et al.

There can be special cases where the values in the input array A vary by $\pm$ 1. In such cases there are different alogrithms to find RMQ with better space bounds. For such cases, Sadakana[20] built an data structure that takes $O(n \log^2 \log n / \log n + |A|)$ space for document retrieval. Fisher and Heun extended this result to improve the space bounds to $O(n \log \log n / \log n)$[15].

2.8 Top-K Retrieval

As mentioned earlier, initial data structures used in this field were the suffix tree and suffix array. However, due to the huge space requirements, these did not do as well as inverted indexes. Its advantages, however, lead it to become popular and in order to bring down the space concerns, compressed indexes were introduced. These indexes include Compressed Suffix Arrays, Compressed Suffix Trees and the FM Index, based on the Burrows Wheelers Transform. Thus, the space taken was competitive with the inverted index but also exploited the advantages of substring and phrase searches. With this compressed indexing became more popular.

As mentioned earlier, document retrieval is an important problem studied. One can answer this problem by finding the documents where the pattern occurs by just solving the Occurrence Listing problem. However, a certain pattern may occur extremely frequently

in a certain document (as each document may be connected with a different subject), but overall may have very few documents that it may occur in ($ndoc$). So, if there could be an index that could retrieve the documents according to how many documents the pattern occurs in, rather than the overall occurrence, it may work faster. Thus, the first framework was given by Matias et al [21], which answered the query in $O(P \log D + ndoc)$time. The algorithm used a generalized suffix tree augmented with extra edges. Muthukrishnan [29], then improved these bounds to $O(P + ndoc)$ by replacing the augmented edges with a divide and conquer approach. The space taken was, however, $O(n \log n)$ bits, which was later improved by Sadakane to $O(|CSA| + 4n + O(D \log n/D) + o(n))$bits[18].

In the IR community and database community, the study of Top-K retrieval is important, as there are many cases where only K documents are required (for example in Web Searches). Bialynicka-Birula and Grossi[7] gave a general framework that can add rank information to items being outputted from any range reporting data structure. Recently, Hon et al [14] came up with a solution that solves this problem in $O(P + K \log K)$ time while keeping $O(n \log n)$ bits of space. This thesis is based on implementation and experimentation on this data structure. A compressed version was also given with $2|CSA| + o(n) + N \log(n/N)$ bits of space and $O(|q| + K \log^{4+e} n)$ query time.

Chapter 3

Index Construction

This chapter goes to explain, in a more in-depth way, the specific data structure we build to perform Top-K retrieval. The index is based on the generalized suffix tree (explained in Chapter 2) and can therefore perform even substring and phrase queries effectively. This data structure takes linear space.

As a base, we build a generalized suffix tree over all documents. The construction of this takes linear time and linear space. Such a tree has N leaf nodes (N being the size of text) and at each leaf node, we store the Document ID of the suffix that branch represents.

3.1 Lowest Common Ancestor Queries

This structure is built on a tree, so as to be able to perform quick Lowest Common Ancestor (LCA) queries. By definition, given two nodes, x and y, we define its lca as the lowest common (node away from the root) ancestor that they both share.

The problem of finding the LCA of two nodes (in a rooted tree, as shown above) is a fairly basic concept in algorithmic graph theory. It is an important and useful structure when it comes to data structures based on trees. It has numerous applications and has been used extensively in our data structure.

In [5], an effective technique to find LCA in a tree was proposed. The Range Minimum Query (RMQ) Problem (introduced in Chapter 2) is quite closely related to the LCA problem. The concept uses both the Cartesian Tree[13] and an RMQ built on the same.

A Cartesian Tree of an array $A[0, ..., n]$ is a rooted binary tree, consisting of a root v, that is labelled with a position i, of the minimum in the array A. There are atmost two subtrees connected to v. The left child of v is the root of the Cartesian Tree of $A[0, .., i-1]$ and the right child is the root of the Cartesian Tree $A[i+1, ..., n]$. In short:

- Root: Minimum element of A at position i.
- Left subtree: Recursively build a Cartesian tree on $A[1], ..., A[i-1]$.
- Right subtree: Recursively build a Cartesian tree on $A[i+1], ..., A[n]$.

The Cartesian Tree does not depend on the exact value of the minimum but only on its position. Thus, Cartesian Tree's are not unique to each array.

For an Array : $A = [40, 1, 30, 10, 20, 5, 40, 15, 35]$ the Cartesian Tree would be as follows:

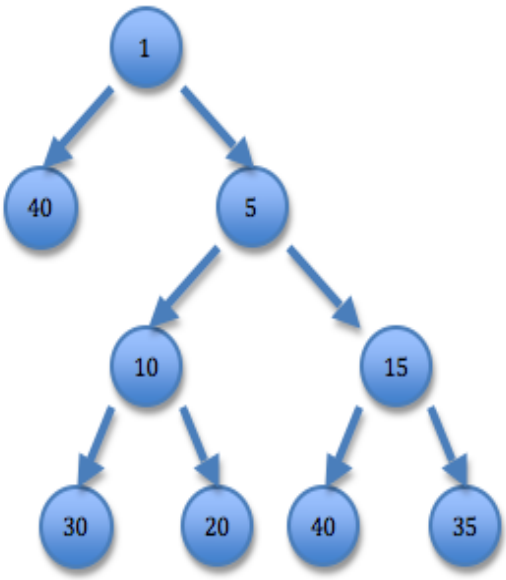


Figure 3.1: Cartesian Tree

The solution to the LCA problem depends on the solution to the RMQ problem. The LCA problem can be reduced to a special case of the RMQ problem based on the observation: The LCA of nodes u and v is the shallowest node encountered between the visits to u and to v during a depth first seach traversal of tree T [5].

- Perform a Euler walk (visit each edge once) in preorder fashion.
- Store the nodes visited during the traversal in an array, EW. This array will contain $2n - 1$ entries.

1	40	1	5	10	30	10	20	10	5	15	40	15	35	15	5	1

- The Level Array of the nodes are then computed. Level Array refers to the level of the node in the Cartesian Tree T.
- As you can see, each element in this Level Array differs from the previous by $\pm$ 1. A specialized RMQ index (as mentioned in Chapter 2) can be built over this.

After building the structure as explained above we can find the LCA between any two nodes in the tree. Suppose we need to find $LCA(30, 40)$ from figure 3.1. From the array in figure 3.1 the nodes between 30 and 40 i.e. $EW[5, ..., 11]$ are the nodes visited when traversing from node 30 to node 40. While travelling to node 40 it reaches the closest point

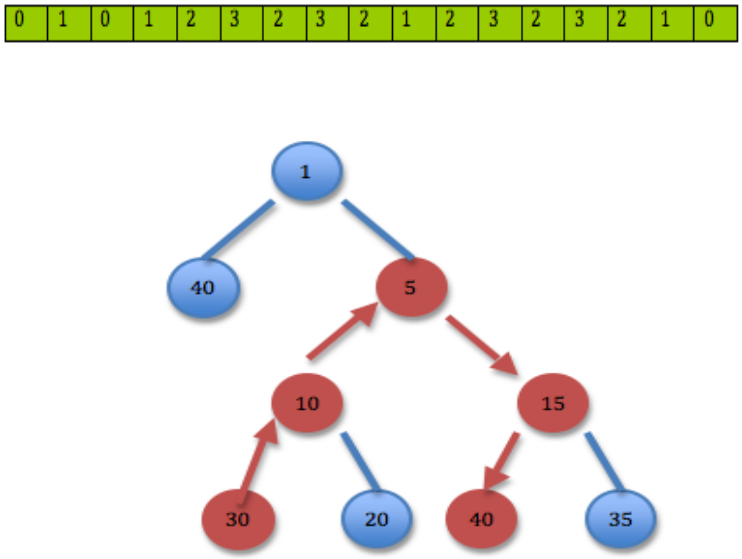
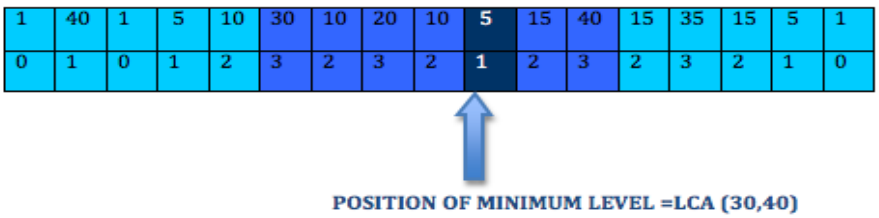


Figure 3.2: Path of traversal between the two nodes 30 and 40, whose LCA we need to find

to 30 from where it can turn to reach 40 i.e.the closest node it has in common to 40 (implying that this is the shortest path it can take through the tree, as shown in figure 3.2).

The highest level it reaches while traversing between these two nodes would be the $LCA(30, 40)$ or the Minimum Level in the $LevelArray[5, ...11]$ would be the common node between 30 and 40. This transforms to a Range Minimum Query between the range (5,..,11).



Thus finding the $RMQ(5, .., 11)$ gives us the answer to $LCA(30, 40)$ which is $EW[min\_pos]$=5 as can be seen in figure. This structure take $O(1)$ time to return the LCA and uses space $O(n)$.

3.2 N-Structure

A structure known as the N-structure was defined in [14]. N-structure is a structure, N_v, which is stored at certain nodes in the GST that has been built. At any node v, we store an array of three tuples, namely document d, frequency c (for a score function of frequency), parent pointer t. Every leaf node will have an entry for document d_i and a frequency of 1. On moving up in the suffix tree, an internal node v will have an entry for document d_i if

and only if at least two of its children have an entry for document d_i in their subtrees. The parent pointer t points to its lowest ancestor which has an entry for document d_i.

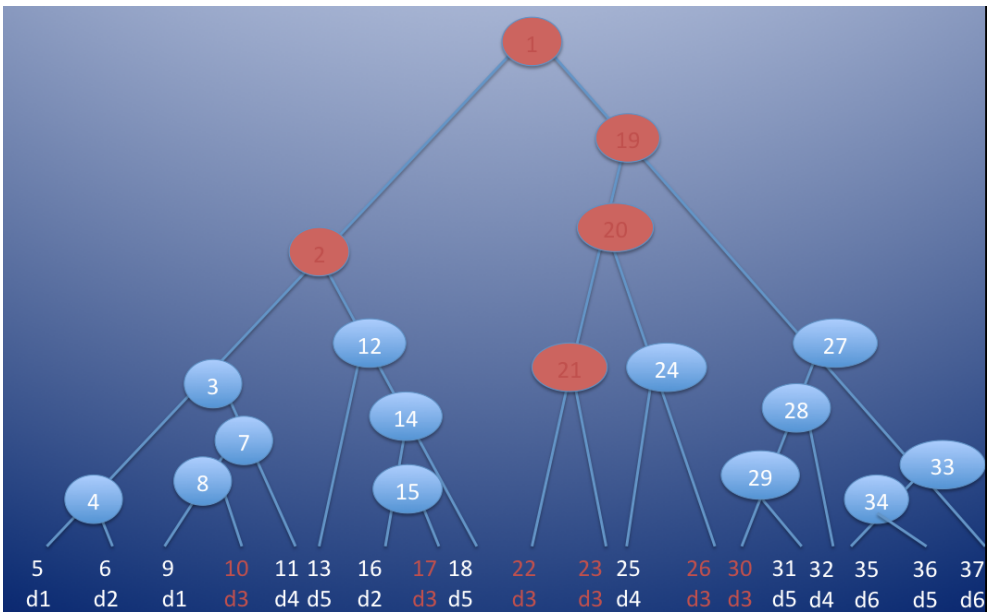


Figure 3.3: Example showing a Generalised Suffix Tree

In figure 3.3, we see a generalised suffix tree where the nodes have been given preorder ranks. To construct the N-structures the leaf nodes, which are appended with the document id that the suffix belongs to, are traversed. The leaf nodes that belong to document d are identified and the lca of every consecutive two leaf nodes are taken. The red nodes in figure 3.3 represent the lca's and the leaf nodes that belong to document 3. A document suffix tree is then formed using the nodes that were identified as lca's and the leaf nodes that belonged to doucment d_3. For example if we take d=3 we get a tree as shown in figure 3.4.

Using this tree we can populate the N-structures for each document in the GST. Every node in this document suffix tree will have an entry for document d_i. The leaf nodes will each have a frequency of 1 and a document id of d_i (which is 3 in figure 3.4). The parent pointer for each of these nodes will be the parent of the node in the document suffix tree. As you go higher in the tree, each node will have the same document id of d_i and parentpointer pointing to its parent in the document suffix tree. The frequency at each node that is not a leaf node is equal to the number of leaf nodes (in the document suffix tree) that fall within its subtree. As you move up the tree this field can be incremented accordingly.

This same procedure is followed for each document at the end of which the N-structures would have been augmented to the GST, as shown in figure 3.5. It is important to note that each node can have one entry per document.

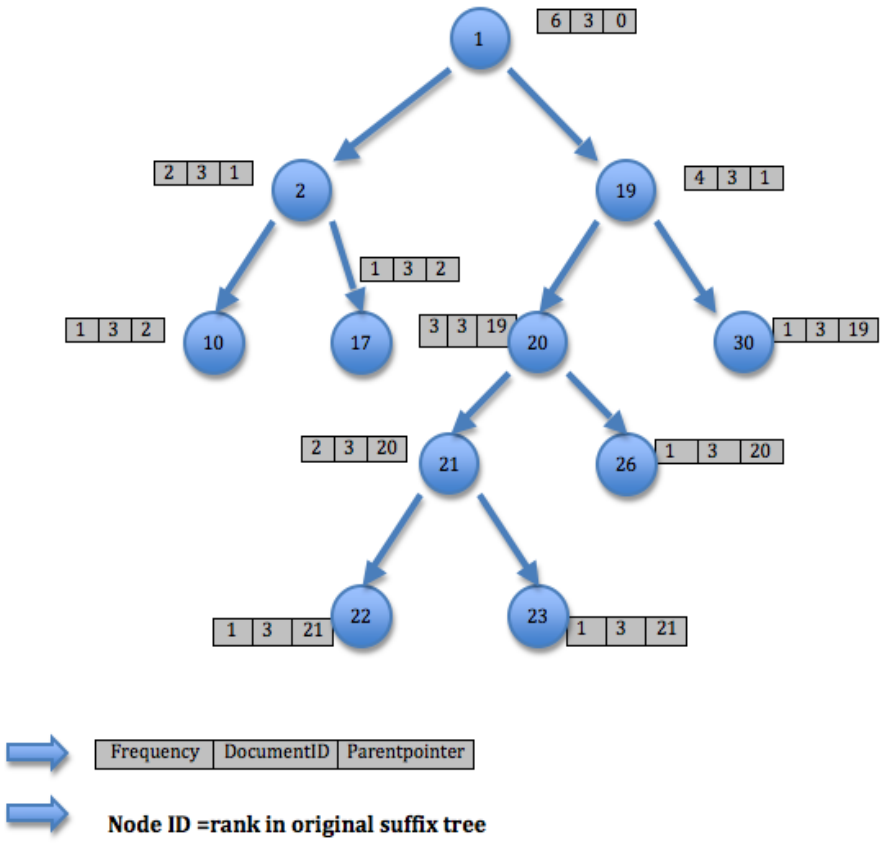


Figure 3.4: Example showing N Structures creation in a Document Suffix Tree ; D=3

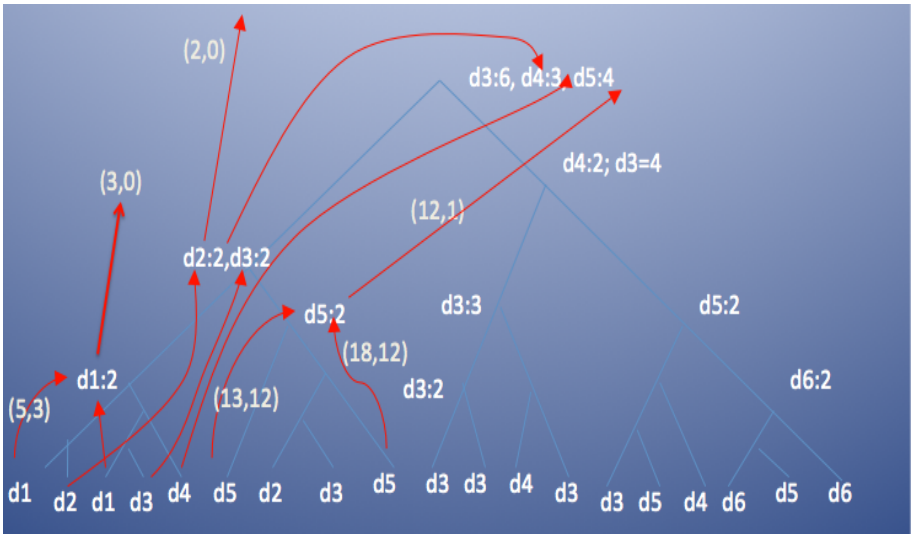


Figure 3.5: Example showing N-Structures creation in a Suffix Tree

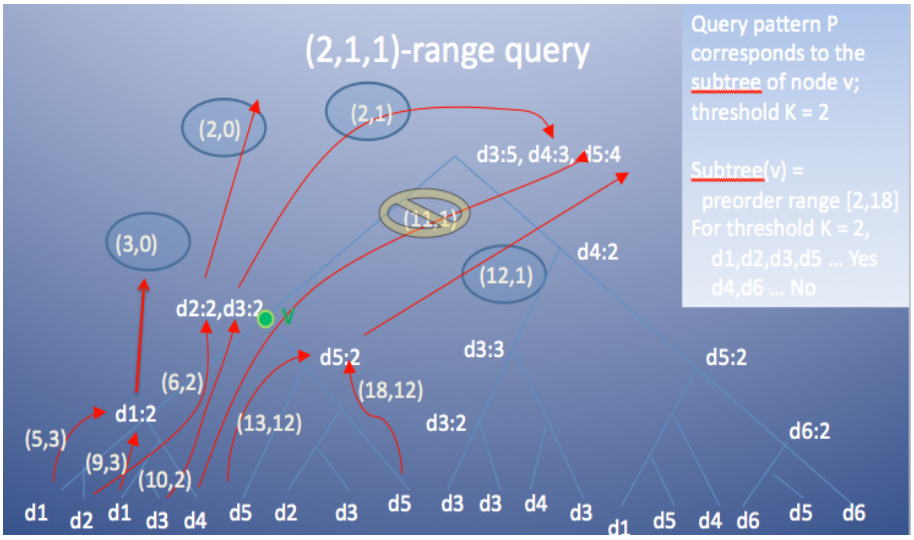


Figure 3.6: Example showing I-Structures creation in a Suffix Tree

3.3 I-Structure

If we take a node v, where the pattern matches, we have a (2,1,1) range query. We first need to find parent pointers that originate from the subtree of node v (which creates two constraints) and these parent pointers must point outside the subtree (to one of the nodes that fall within the path of the pattern) to ensure that only distinct documents are obtained as shown in figure 3.6. Finally, out of these entries the Top-K documents, according to score function (frequency in this case) need to be extracted. Since a (2,1,1) range query is difficult another entry known as the I-structure is introduced.

One of the fields in the N-structure stored is the parent pointer. For any node v, which has an entry for document d_i, the parent pointer t points to a node x. At node x we store an array of three tuples, namely frequency, document id and origin. The origin refers to the pre-order rank of the node v whose parent pointer pointed to node x, the frequency stores the frequency at node v and document is d_i.

For document d=3 the document suffix tree is shown in figure 3.7. The entries at every parent pointer now store the information of its children (in the document suffix tree). I-structures can be constructed for each document based on the N-structures.

The structure is stored in increasing order of the origin entry (pre order ranks). If there are entries coming from the same node (multiple entries with same origin value), they are stored in increasing order of documents. Entries with same document value but from different origins can also exist. On the array containing the frequency values an RMQ strcuture is built to find the maximum value, quickly, given a range.

One important observation that can be made in figure 3.7 is that from each subtree we get only one entry per document. Therefore, if the pattern matched at node 20 there is only one I-structure entry from the subtree [20,26] in any node from the path between node 20 and root that has an entry for document 3. We can, however, have entries from the same subtree but for a different document like, for example, at node 2 we have entries from

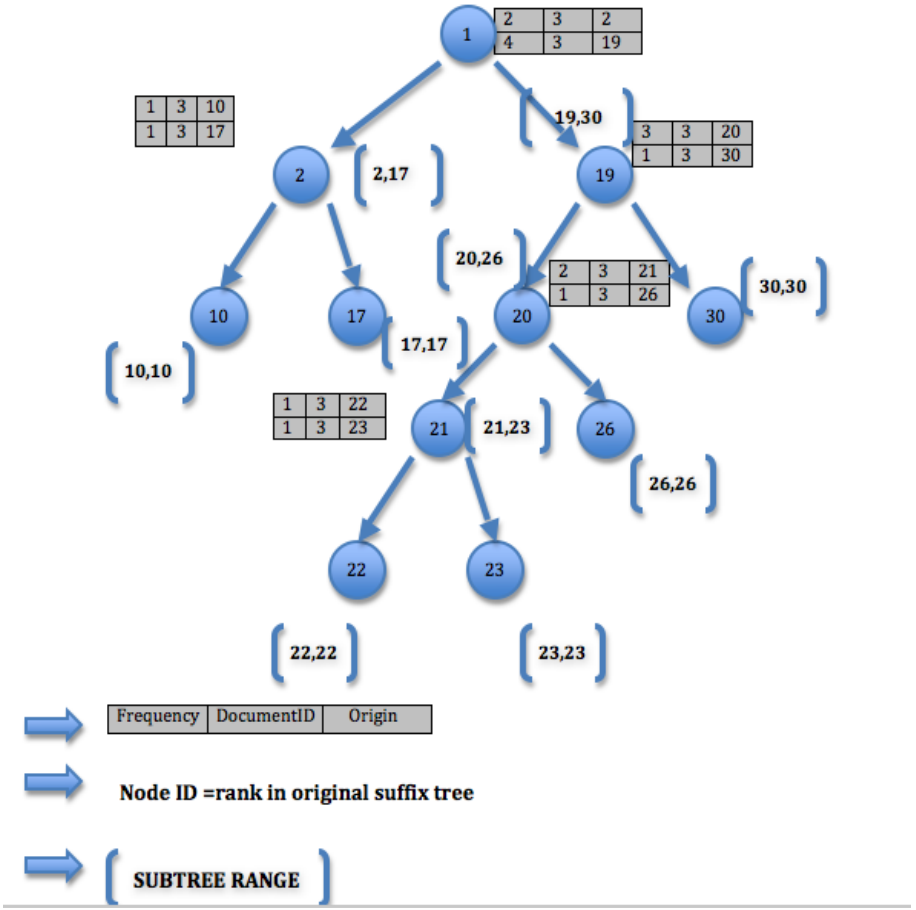


Figure 3.7: Example showing I Structures creation in a Document Suffix Tree: D=3

document 3 and from document 2 from the sub tree [3,11]. Once the I-structures are created the N-structures can be deleted.

3.3.1 Variation on I-structure

Using this index, our implementation also provides a scheme to return the Top-K documents according to their page rank value. Thus, instead of storing the frequency we simply store the page rank value of the document and perform the same technique to retrieve the documents. This index can also be extrapolated to retrieve Top-K documents for two pattern queries according to their page rank values.

3.4 Querying The I-Structure

Let us assume that we are searching for the Top-K documents in which a pattern P, occurs. To find documents that P occurs in, we search the I-structures in all nodes that exist in the path from root to the node where the pattern matched. The I-structures have three parameters, Origin, Document ID and Frequency. The I-structures whose origin are from within $P's$ subtree need to be identified (which can be done by performing a binary search over the sorted origin array) as shown in figure 3.8. The frequency of these entires would give us the frequency of P in the document.

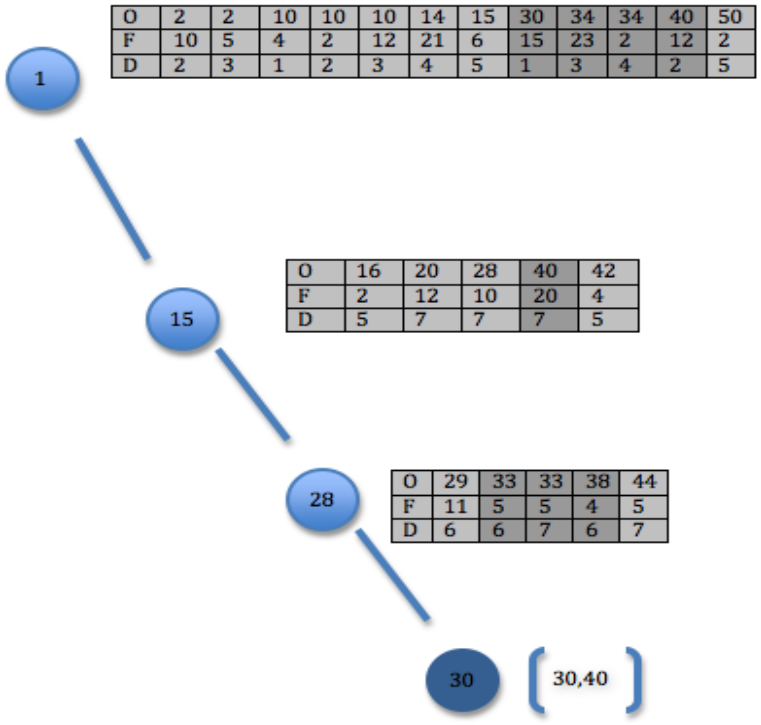


Figure 3.8: Example showing the retrieval of Top-K documents

A Range Maximum Query data structure, built on the frequency field, is used to find the position of the maximum frequency, at each of the $|P|$ nodes that go from root to node where P matched. Each of these maximum values is inserted into a heap as shown in figure 3.9.

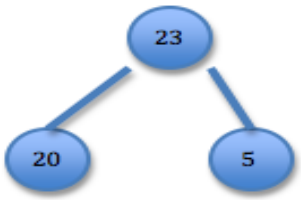


Figure 3.9: Example showing the heap after traversing from P to root

After inserting the maximum values from each range, we extract the maximum value from the heap and identify the range, for which this value was maximum. The range of I-structures at the identified node then gets divided into two parts and we find the maximum value within each of the two ranges and insert into the heap. We recursively continue to perform the same function (extract max from heap, identify the range it was taken from, break the range at the position of this value,find maximum in each range and insert into heap) till we have extracted K values form the heap. At this point we will have the the Top-K documents that our pattern occurred in and the frequency with which it occured in each. In figure 3.10 when Top-1 is extracted i.e. 23 the range that 23 was taken from is identified (at node 1 (8,11)). The range is then divided and from range (10,11) we obtain the maximum, 12 and from range (8,8) the maximum 15 and insert these values into the heap as shown.

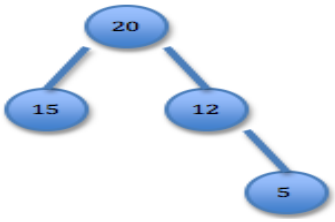


Figure 3.10: Example showing the heap after extracting Top-1

This index can be used to perform document listing and output all the I-structure entries from root to node P whose origins fall within the subtree of P. Comparative studies are shown in Chapter 5.

Chapter 4

Implementation

The complete index along with suffix tree can take upto 70 times text. To provide a space efficient index to retrieve the Top-K documents the index is compressed by using the following techniques.

4.1 Compression of Trees

When we consider a tree data structure one of the biggest bottlenecks for this data structure is the space it occupies (especially a suffix tree). There are three ways in which this can be done, Balanced Parantheses Encoding, DFUDS and LOUDS.

We need to be able to traverse the tree. To be able to do this we need to store pointers at each node. Each pointer needs atleast $\log n$ bits of space to be able to address n different locations (n being the number of nodes in a tree). This leads us to have to store $O(n \log n)$ bits to represent a tree. The more number of operations we need this tree to perform (like finding parent node) the higher the constant becomes. Instead, by using compression, we can store these trees in $O(n)$ bits and provide structures to perform operations to perform effective tree traversals. Basic operations to provide for navigation functionalities are as follows [4]:

- $findclose(x)$: Find the index of the closing paranthesis that matches the opening paranthesis at position x.
- $findopen(x)$: Find the index of the opening paranthesis that matches the closing paranthesis at position x. Between its opening and closing paranthesis lies the node information.
- $enclose(x)$: Find the index of the opening paranthesis of the pair that most tightly encloses node x.
- $rank_{(}(x)$: Find the number of opening paranthesis upto position x.
- $rank_{)}(x)$: Find the number of closing paranthesis upto postion x.
- $select_{(}(x)$: Find the position of the x^{th} opening paranthesis.

- $select_{)}(x)$: Find the position of the x^{th} closing paranthesis.

4.1.1 Balanced Parantheses

A tree with N nodes can be represented by a string of length $2N$, using this representation. Each node is represented by a pair of opening and closing paranthesis (), thus leading to there being N left and N right bracket sequences. The subtree of the node is represented in between each nodes opening and closing bracket [(.....)]. The code is built by performing a depth-first search on the tree. Opening paranthesis is written when you arrive at the node for the first time and a closing paranthesis is written when you leave. An example of this representation for figure 4.1 would be, (((()(()()))()())). As can be seen there are 8 opening paranthesis and 8 closing paranthesis (balanced). A leaf node can be identified by the consecutive opening and closing sequence, ((()(()()))()())

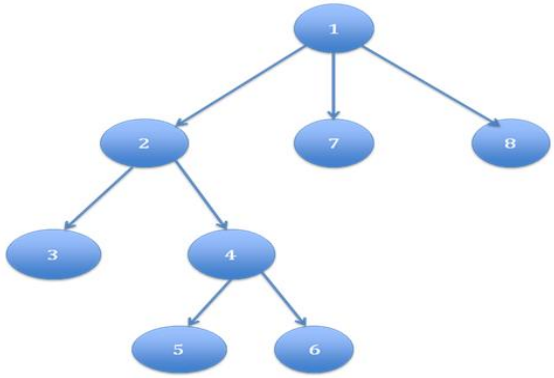


Figure 4.1: A tree structure marked with preorder ranks

Using this sequence the tree can be recreated, but, we need to be able to navigate through the tree for it to be useful. If we take each opening bracket representation as '1' and each closing bracket as a '0' we have ourselves a bitstream of zeros and ones. To be able to perform navigational operations we need to build an additional data structure of $o(N)$ bits. The basic navigational operations can be brought down to the operations shown above. Hashing based Heuristics [23] can be used to store the values of findclose/findopen. A three level blocking is shown to be optimal, where we take two parameters, $b = \log N$ and $s = b \log N$. Values for distance less than b are calculated by using a stored excess array (how many opening paranthesis exist more than closed paranthesis, upto this position). For distances between $b + 1$ and s distance from the position are stored in $\log s$ bits. For greater distances the actual value is stored in $\log n$ bits.

Using this we can perform all basic navigational operations like $parent(x)$, $firstchild(x)$, $sibling(x)$, $degree(x)$. We can also find the $LCA(a, b)$, the $rank(x)$: preorder rank of x, $select(x)$: position of node with rank x and other such important operations. The i^{th}

child and $degree(x)$ operations are performed by finding the $firstchild(x)$ and then tracing through its siblings. Compressed Suffix Trees (CST's) uses the balanced paranthesis representation for the tree as lca operations are more frequent and important for suffix trees (which DFUDS could not incorporate). In our implementation we use the Compressed Suffix Tree(CST) to construct our index and to perform pattern match(simulating the suffix tree which would take more space than the CST would).

4.2 Encoding Techniques for Numeric Arrays

The I-structures occupy the next big chunk of space in our index. There are different encoding schemes available to us to encode each field of the I-structure (Origin, Document ID and Frequency).

1. VARIABLE LENGTH ENCODING Normally, while programming, we store every number in an array in the form of an integer array. This would imply that it stores each value using 32 bits. However, some numbers can be as small as 2, which needs only 2 bits to represent it. The most space efficient way to store this array would be if each number only took as many bits as was needed to represent it. Therefore, if each number is represented using only $\log i$ bits, where i is the value, we would have very effective compression. Since each number may take different number of bits, this is called variable length encoding. To find the number we are looking for, we can store a bit vector (a vector of zeros and ones) over this array, where 1 represents the beginning of a new number and 0 represents continuation of the number. Thus if we representing numbers A=[2 10 50], we can represent it as A_{enc}=[10 1010 110010] and $bitvector$=1010001000001. This though leads us to waste an equivalent number of bits on the bitvector and an additonal structure should be built on the bit vector to be able to retrieve the values.

   To retrieve the values back we need to build an additional structure on this to perform select operations. $select(i)$ returns the position where the i^{th} 1 is present. Thus, in the above example, if you wanted to retrieve the 3^{rd} number $select(3)$ would give 6. The bits in A from 6 to $select(4)$, which is at position 12, give you the number 50. The index to perform rank and select queries can take $nH_0(S) + o(n)$, where $H_0(S)$ represents the zeroth order empirical entropy of S to perform queries in O(1) time.

2. FIXED LENGTH ENCODING As described above, variable length encoding may not be an optimal solution, even though it seems most likely. Another variation would be to find out the maximum value in the array we are compressing and store all values with log(max) bits. This will work best if the biggest number requires way less than 32 bits and all numbers need around the same number of bits. The advantage in this method is that we do not need to store a bitvector or a rank/select structure above. This may save significant space and time (while decoding the values).

   To implement the above we will have to make use of bit shifting operations as standardised data types available are only 8, 16, 32 etc bits (unless the max number is a multiple of 8).

3. GAP ENCODING This encoding scheme is applied when the array is sorted. It would work well if the difference between consecutive elements is not very large. Thus, instead of storing the explicit values we store its difference with the previous number (which would require lesser number of bits as the difference should be small). A bit vector can be stored above this to decode the values. As an example:

   A=[1 3 4 5 8 14 16 20]

   A_{enc}=[1 2 1 1 3 6 2 4]

   The encoded values are smaller than the original array and therefore will take fewer number of bits to store. The differences have been stored using variable encoding scheme explained earlier in this section.

   This has a disadvantage of having to go through all previous values to find the value we require. If we need to decode the above encoded array and find the 5^{th} element we need to add every element in the array, upto the idex of the value we need, to get the value (2+1+1+3+6+2+4=20). This can create a huge time overhead. One way to reduce the time overhead would be to implement a blocking system and store explicit values at the beginning of each block. Another variation would be not to store the consecutive difference, but the difference with a value at the beginning of the block (thus preventing the overhead of going through the whole block to decode the value). The disadvantage for this would be that space would be higher. Another variation is that the consecutive differences can be stored along with explicit values at certain intervals. This would also refer to a blocking scheme where you only need to walk within your block to decode the value. Many time space tradeoff's can be worked out in this way. We can also play around with the blocking scheme and implement either fixed blocking or variable blocking schemes.

4. RUN-LENGTH ENCODING Sometimes there may be a run of the same number in some arrays. In cases like this, it seems unnecessary to write each of these values explicitly (even if delta encoding has been applied to it, it can lead to a run of zeros). Thus, it may make more sense to write how many times the number repeats rather than repeat the number, so as to save space. This would also prove beneficial when performing operations like binary search on this array (as the size of array it needs to go through is significantly reduced).

Each field in the I-structure has a property that enables us to compress them effectively.

4.2.1 Origin Compression

As mentioned before the Origin's are stored in increasing sorted order. We also observed that entries come from node's that are quite close together. Thus the difference between consecutive entries are relatively smaller. Using these properties gap encoding seems to be the most promising method of compression. We store explicit values at every b^{th} entry, b being the blocking factor. However, the time taken to walk through the array to decode and the space taken by the explicit values are major disadvantages that this method has.

Another observation was that many of the entries come from the same node thus leading to a run of entries having the same origin value. Thus another encoding technique that we applied was run length encoding where we simply stored how many times the number repeated rather than storing the number that many times. On this encoded array gap encoding could be applied.

Another technique that we performed to compress the origin values was to store the difference between the origin entry and the pre-order rank of the node the I-structure represents. This also led to considerable compression and beats the two disadvantages with gap encoding, having to walk through the entire block to decode the value and storing a very space expensive explicit value. Another variation that can be applied is to store the explicit values as the difference between the origin entry and the pre-order rank of the node. Run length encoding was applied on this scheme as well.

When it comes to blocking there are two different schemes that can be applied:

1. Fixed Blocking Scheme We consider a fixed size as the blocking factor and store explicit values for every b values.
   O=[2 2 10 10 10 14 15 30 34 34 40 50]
   $EncArray$=[2 0 8 0 10 4 1 15 34 0 6 10]

2. Variable Blocking Scheme We decide to store explicit values only at entries whose difference from the previous value is higher than a set threshold. This may appear to be beneficial but requires an extra bit vector to be stored defining the boundaries where the explicit values are stored, thus increasing the space taken.
   $EncArray$=[2 0 10 0 0 4 1 30 4 0 6 50] for a threshold of 8
   A comparison between these schemes is provided in Chapter 5 in section 5.4.5.

4.2.2 Frequency Compression

We notice that most entries in the frequency field are very small values (about 95% $\leq$ 2). Thus we form a four tier system to store the values effectively. We create four arrays that store 4 bit, 8 bit, 16 bit and 32 bit values each. Each entry is pushed into one of these arrays according to how many bits they take and a bitvector is built on them to accertain which array to pick the value from. This proves to be an effective form of compression as most entries fall in the lower bracket and take just as many bits as they should and thus save about 28 bits each.
For example, if we have a frequency array F=[1 1 1 2 7 3 23 54 1 3 18 2 14 80 20 95000]
Size =16*4 =64 bytes
$Array1$=[1 1 1 2 7 3 1 3 2 14] =40 bits
$Array2$=[23 18 20] =24 bits
$Array3$=[54 80] =32 bits
$Array4$=[95000] =32 bits
Total size= 16 bytes

4.2.3 Document Compression

Since each value from 1 to d will be present as an entry in the document array, the only way to compress this is to make sure that they only occupy as much space as the largest number needs. We thus use Fixed Length Encoding and store each value using $\log D$ bits where D is the number of documents present in the text(or the largest value that will need to be stored).
For example, if D=[3 10 5 13 25 64 2 7 25 31 70 100 32 53 77 39]
Size =16*4 =64 bytes
Assume we have 100 documents; 100 takes 7 bits
D_{enc}=[0000011 0001010 0000101 .....]
Size =(7*16)/8= 14 bytes

4.2.4 Page Rank

We give each document a unique random number from 1 to D, where D is the number of documents, to simulate a page ranking system. These values follow the same trend as the document array and thus are stored using the same Fixed Block Encoding scheme using $\log D$ bits.

4.3 Two Pattern Queries

We can have a situation wherein the user may want to find the Top-K documents on giving two patterns as input. Using our index to retrieve Top-K documents for one document we have extended the same to perform two pattern queries.

Let us say we have two patterns $P1$ and $P2$ that match at nodes $p1$ and $p2$. We can seperately find the Top-K documents that each of these patterns $P1$ and $P2$ occur in. Once we have two lists of Top-K documents for each pattern, we need to perform certain heuristics to retrieve the highest page ranked documents that both the patterns occur in. The documents come out in page rank order. We traverse through the list of Top-K documents for pattern, $P1$, and find in the corresponding list for pattern, $P2$, if this document exists. Since the page ranks are sorted in order and each document has an unique page rank, if the list of pattern, $P2$ has this document its page rank would also be present and can be found by performing a binary search over the page rank's of pattern $P2$. Before we perform the binary search, we need to determine which of the following cases our search will fall under (list 1 is the list which contains the documents in which pattern $P1$ is present and list 2 is the same for pattern $P2$)

- Check if each list occurs in atleast K documents. If either of these patterns occur in less than K documents we retrieve only as many documents as the pattern occurs in and traverse the list that occurs in less than K documents. Once we have traversed the entire list for the pattern (list 1 or list 2), we finish even if we do not have K documents
- Check if the lowest page rank in list 2 has a page rank that is higher than the value we are searching for. In this case, we know that our document cannot be within the

Top-K and generate the next $K+1$ to $2K$ documents for list 2 and perform this check again.

- Check if the highest page rank in list 2 has a page rank which is lower than the value we are looking for. In this case, we first make sure that the lowest value in list 1 is not greater than the highest value in list 1. In case this is true, as in step 2, the next K documents from list 1 are retrieved and the check is performed again.

- If all the above checks have been performed and they came out false, we traverse either list and perform a binary search in the other list to find if the page rank exists there. If it does then we can directly extract it out.

- These steps are performed till we get K such documents by generating the next K documents if we exhaust the list.

4.4 Phrase Searching

Using our index to generate Top-K we also make an alteration to the index to create a specialized index to produce Top-K documents where the queries are phrases. In suffix tree based indexes, one of the advantages is that we can find the occurence of any substring as it would appear as a suffix in the tree. For example, in 2.1 we can not only find BANANA\$ we can also find ANANA\$ or AN if we needed to. However, if we are sure that we are never going to get substring queries, we can create a word based suffix tree like in figure 2.2. This avoids having to store substrings and the other auxillary strucutres related to them like the I strucutres for these substrings. The tree also becomes more sparse and the space spent encoding it and the time take to traverse these auxillary structures reduces. In essence, we keep information only for strings that begin after a space(" ") or any other word separating character leading to a heavy drop in space. This brings it down to a structure comparable to the Inverted Index as just like the Inverted Index we keep only words and their Document ID's along with some auxillary structures and we have the added advantage of not having to go through a position lists to find words that are located next to each other (for phrase queries) as we can match the entire phrase directly using a suffix tree. Experimental results for this are provided in Chapter 5.

To create a specialized phrase searching index we do the following:

1. Traverse the edges of the suffix tree. If the edge has a space or any word seperating character we mark the end node of the edge. These are the only nodes of the suffix tree we take into consideration for the phrase index. We can simplify this traversal by identifying the child whose first_character is space(which will most likely be the first child as the ascii value of space is 32 and is most likely to lexicographically be the smallest). All the phrases that may exist in the index will definetely be a child of this node. This creates one restriction that the query must always start with a space.

2. Once the nodes of the new reduced suffix tree have been identified we can create the N-structures and I-structures based on this new suffix tree.

3. When a phrase query is given as input we identify the Top-K documents in the same manner as we do for regular queries.

In section 5.4.8, this phrase index is compared to an inverted index to retrieve the Top-K documents in which a query pattern which is a phrase occurs. An inverted index performs phrase searching in the following way:

1. For each word in the phrase a document list (of all the documents the word occurs in) is generated.

2. An intersection of the document lists of all the words should be found.

3. The position list must be traversed to find only the documents where each of these words occur next to each other in correct order.

4. The Top-K list must then be generated out of the documents that cross all three levels.

Chapter 5

Experimental Analyses

In this chapter, we provide an empirical study and analyses on the practicality of the index. We explain the experimental setup and provide variations and extended functionalities that the index possess. We also study the space taken by the index and provide alternate methods thus creating an opportunity to take advantage of different tradeoff schemes.

5.1 Experimental Setup

We primarily use two data sets based on which we perform the experiments.

- Random : The text in this collection has been generated randomly containing a mixture of different english texts(e.g. bible), whose alphabet size Σ=26 and contains only letters from a to z. The total size of this collection comes upto 6MB with 1600 documents.
- Enron Email Dataset: The text in this collection is a collection of emails sent within enron. Source of this dataset is : http://www.cs.cmu.edu/ enron/. This colelction contain alphabets, numbers and symbols and the total size of the collection comes upto 16 MB with 5060 documents.

5.2 Platform

This index was implemented using the programming language C++ with a GCC 4.2 compiler. The operating system is Mac OS X 10.6 with a 64 bit architecture. The processor is 2.26 GHz Intel Core 2 Duo with 4 GB RAM.

5.3 Source Code

Components of the Source code both for implementation and comparitive studies were obtained from the following sources:

- Compressed Suffix Tree : We used the implementation(based on Sadakane's method) available for download at http://www.cs.helsinki.fi/group/suds/cst. This implemen-

tation is a part of the SUDS project at the University of Helsinki Alternatively, code for Compressed Suffix Tree can also be downloaded at http://www.uni-ulm.de/in/theo/research/sdsl.htm

- Bit vector Dictionaries : We used the implementation available for download at http://www.uni-ulm.de/in/theo/research/sdsl.htm. Compiler options -O3 and -DNDEBUG must be used to get best performance.

5.4 Experimentation

5.4.1 Experiment 1

The theoretical bounds provided by the paper [14] for retrieval of Top-K documents are $O(P+K \log K)$. In our first experiment we check the time taken to retrieve Top-K documents practically. We use the Compressed Suffix Tree here to match the given pattern and return the range $[sp, ep]$. The I strutures in this experiment are uncompressed.

We also show the variation of retrieval time as a function of K. Experiments have been performed using data set 2, with 10 patterns for each length and running each query 20 times. In figure 5.1 we also show the variation of retrieval time with pattern length.

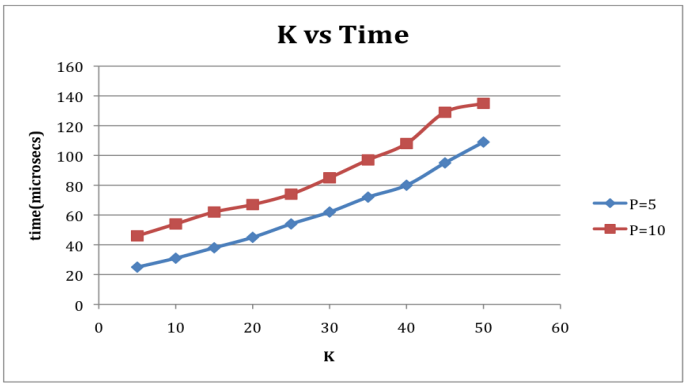


Figure 5.1: Shows the effect of varying K on the retrieval time (for patterns of size 5 and 10 showing variation of $|P|$ with time)

5.4.2 Experiment 2

We now show the time taken to retrieve the Top-K documents when the I-structures are compressed (explained in section 4.2). The set up remains the same. Data Set 2 was used for these experiments.

Just as Experiment 1, in figure 5.2, we show that the retrieval time increases with increase in K. We show the increase in retrieval time for patterns $|P|$=10 as compared to $|P|$=5 in figure 5.2.

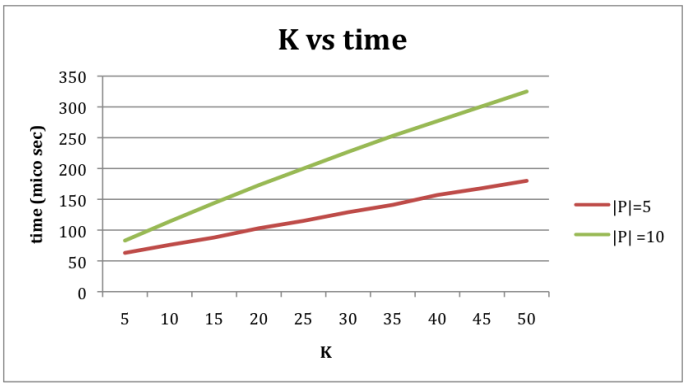


Figure 5.2: Shows the effect of varying K on the retrieval time

5.4.3 Experiment 3

We show the variation in retrieval time on compressing the I-structures. This experiment shows the effect of compression (as explained in section 4.2) on the retrieval time. Data Set 2 was used for these experiments.

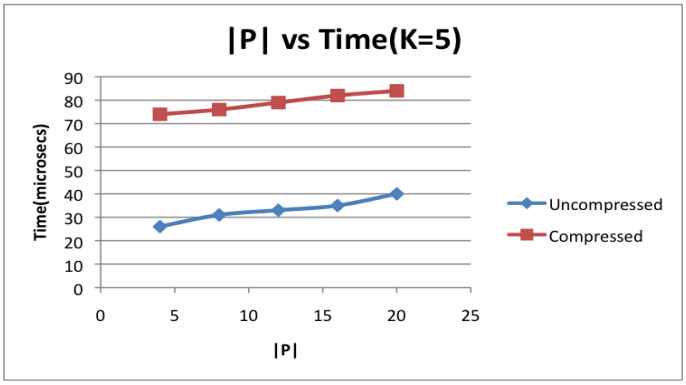


Figure 5.3: Shows the effect of varying $|P|$ on the retrieval time

Both figures, 5.3 and 5.4 show an increase in retrieval time on using compressed I Structures. Compressed Indexes will take some more time to retrieve the documents as they need extra time to decode the encoded values. We see that the retrieval time for the compressed version is approximately doubled.

5.4.4 Experiment 4

We study the space taken by the index, both in its compressed form and its uncompressed form. We can see the space taken by each component in figure 5.5 and 5.6. Data set 2 was used for these experiments.

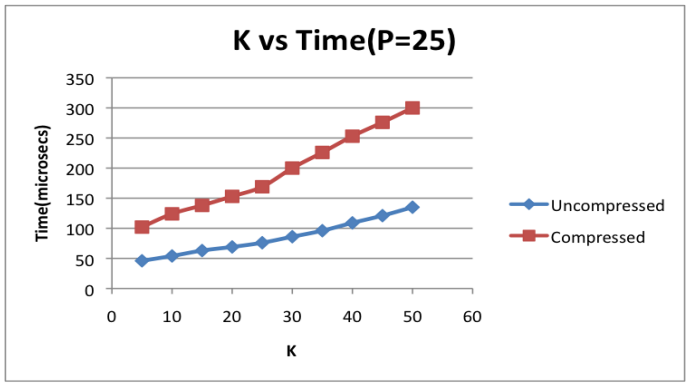


Figure 5.4: Shows the effect of varying K on the retrieval time(Experiment 3)

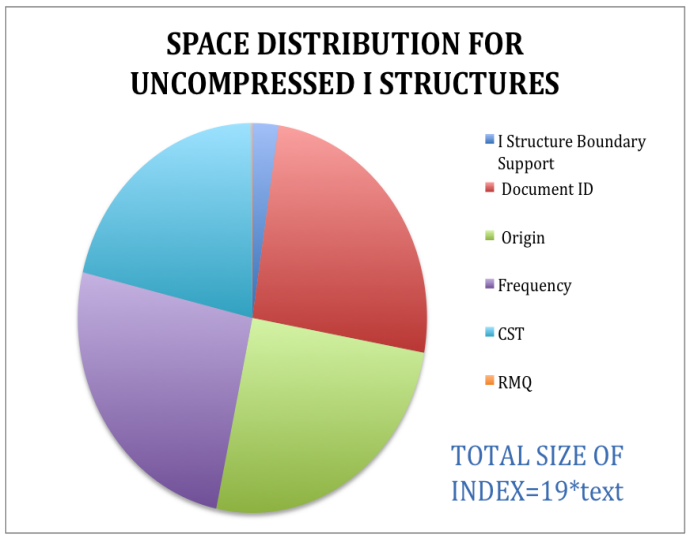


Figure 5.5: Shows the distribution of space in the uncompressed index

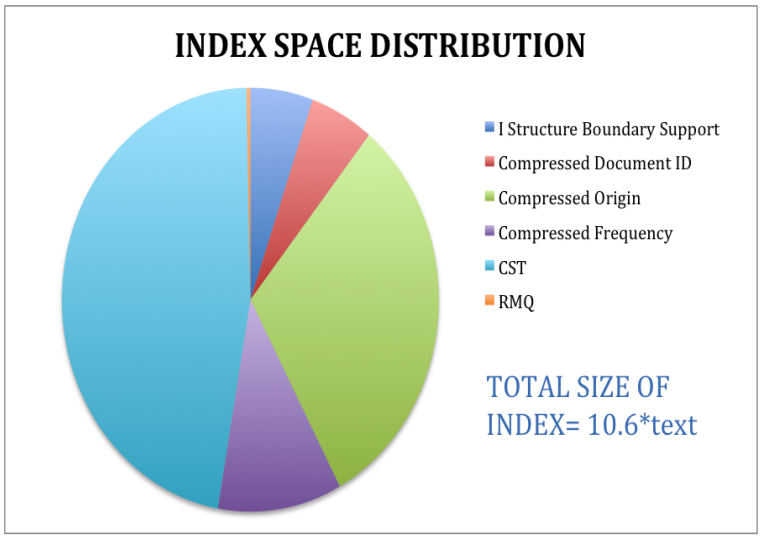


Figure 5.6: Shows the distribution of space in the compressed index

5.4.5 Experiment 5

We study the effect of different encoding methods on the origin field of the I-structures as explained in section 4.2.1. We also show the effectiveness of the compression techniques listed in section 4.2 for all three fields of the I-structures. Data set 2 was used for these experiments.

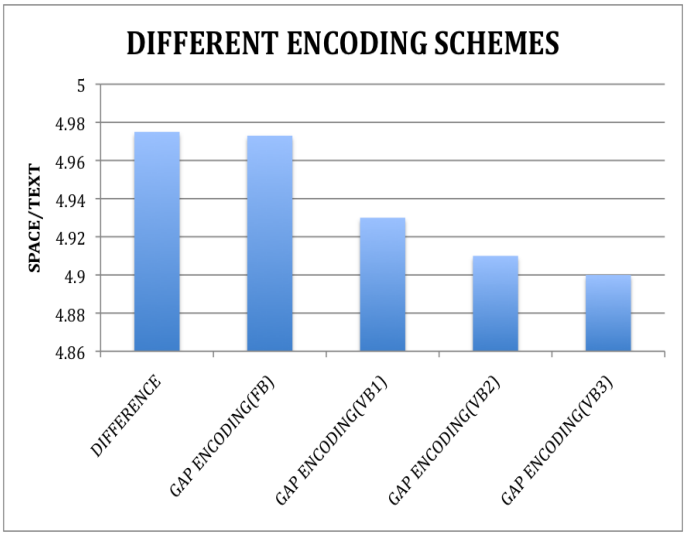


Figure 5.7: Shows the effect of different compression techniques on the origin field on space

After applying the listed compression techniques(in Chapter 4) the level of compression we were able to obtain on the I-structures are shown in figure 5.9.

As mentioned in Chapter 4, there are few techniques that can be used to compress the

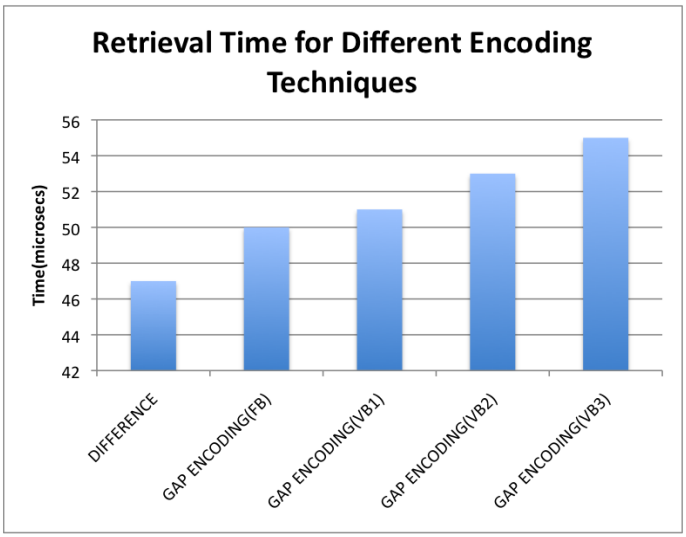


Figure 5.8: Shows the effect of different compression techniques on the origin field on time

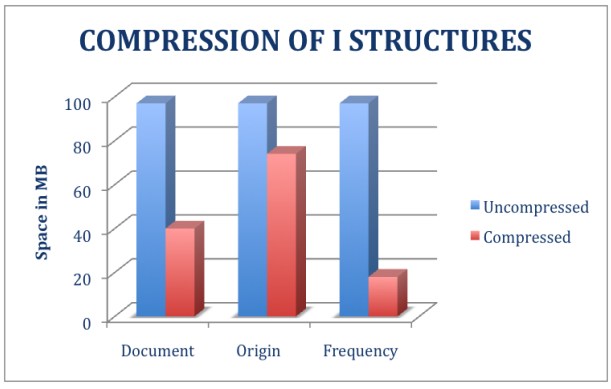


Figure 5.9: Shows the effect of compression on space occupied by I-Structures.

origin field in the I-structures:

1. Gap Encoding FB(Delta Encoding): We store the difference between consecutive values. To prevent having to walk through the entire array (as mentioned in Chapter 4) we store explicit values at certain intervals. Thus the entire array is divided into a number of block. To decode a value, you need to find which block it falls under (in the case of fixed blocking you can simply find $mod(decode\_position/blockingfactor)$ and walk within the block to find the original value. In figure 5.7 and figure 5.8, we use a fixed blocking factor of 10. FB stands for Fixed Blocking.

2. Gap Encoding VB: The essence of this method is the same as above. However, instead of storing explicit values at fixed intervals we store them at variable intervals (VB: Variable Blocking). The Blocking is determined by a difference, only if the difference between the next explicit value and the previous one is above a certain threshold do we store it as an explicit value. In this case, since we cannot directly calculate the block the value we need to decode may occur in, we store a bitvector over the array(1 represents that the value is an explicit value and 0 represents that it is a vlue within a block). The block can now be identified by finding how many one's are in front of it or $rank(decode\_position)$. Once the block is identified, we walk within the block to decode the value as in the case of fixed blocking.

    - VB1: The difference between the explicit values must atleast be 8.
    - VB2: The difference between the explicit values must atleast be 16.
    - VB3: The difference between the explicit values must atleast be 32.

3. Difference: In all the above methods we need to walk a certain distance, within the block, to decode the value. This proves to be time consuming. We therefore adapted a technique wherein we store the difference between the origin value and the rank of the node these I-structures are for. We find, just as the difference between consecutive values in the origin is very small, the difference with the node id is also small. This also has the added advantage of not having to walk to decode the value thus saving some query time.

Run Length encoding was applied on all the above techniques. The space analysis for these techniques is shown in figure 5.7.

5.4.6 Experiment 6

We compare the time taken to perform document listing, by our index, and then retrieving the Top-K documents to directly using the index to retrieve the Top-K documents. Data set 2 was used for this experiment.

This index can also be used to list all the documents that the query pattern occurs in. After pattern matching we find the node P where the pattern mathces. To retrieve the documents that the query occurs in we traverse the suffix tree from root node to node P, just as how we retrieve top-K values. At each node we find the range of I-structures whose origin's fall within the subtree of P and output the document field of these I-structures as the documents that this pattern occurs in.

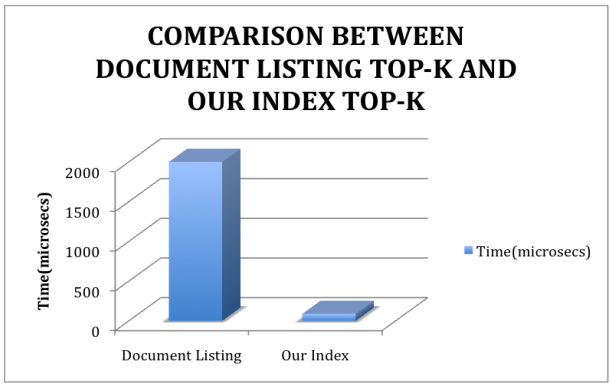


Figure 5.10: Shows the effect on retrievel time on performing Top-K using the document listing approach as opposed to our index which takes advantage of the property that K is mostly probably small

5.4.7 Experiment 7

We study the retrieval of Top-K documents(based on page rank) when 2 patterns are given as queries (both time and space). Data set 2 was used for these experiments. We take two cases to study this behaviour. Explanation is given in section 4.3.

The Two Patterns are highly co-related

Both patterns used in this case are highly co-related and occur in more than 5000 documents together. We show variation of retrieval time for different values of K and patterns of different length.

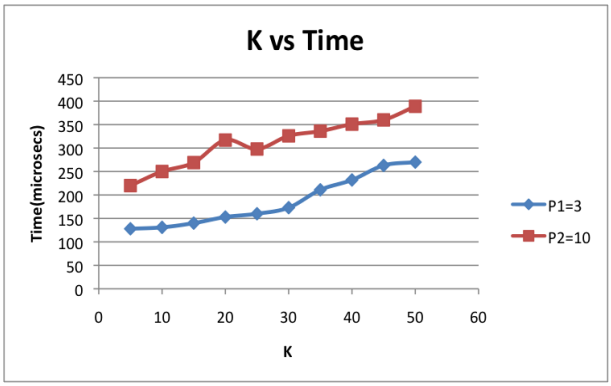


Figure 5.11: Shows the effect of varying K on the retrieval time

The Two Patterns are not very highly co-related

For Pattern $|P|$=3, $P1$ occurs in 534 documents and $P2$ occurs in 2000 documents. For Pattern $|P|$=10, $P2$ occurs in 930 documents and $P2$ occurs in 1500 documents. From figure

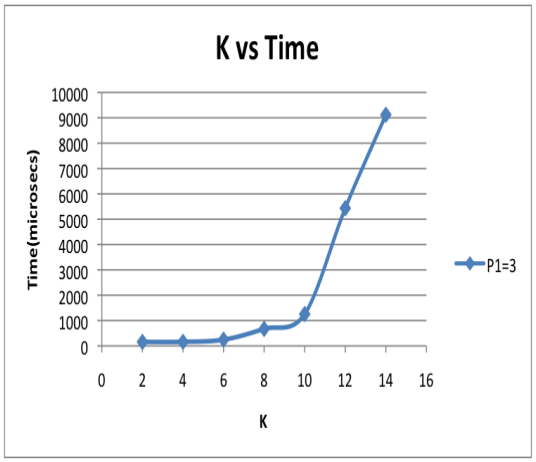


Figure 5.12: Number of Common documents =13 resulting in the heavy rise in retrieval time after K=12.

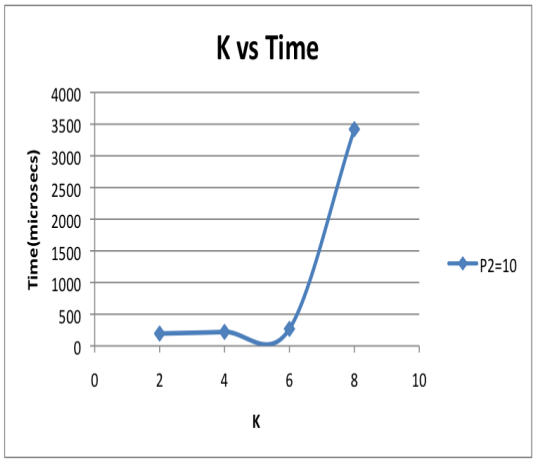


Figure 5.13: Number of common documents=6 resulting in the heavy rise in retrieval time after K=6.

5.13, we can see that since P=10 must go through atleast 930 documents, as compared to P=3, which has to go through 534 documents, to accertain that there are no more common documents the increase in retrieval time at K=8 for P=10 is more than at K=14 for P=3.

5.4.8 Experiment 8

We study the retrieval of Top-K documents when the queries are phrases. We study both the space and time to retrieve these documents and compare them to the Inverted Index. We use data set 1 for this experiment. Explanation of the phrase index is given in section 4.4.

Space Analysis

As mentioned in Chapter 4 we extended our Index to provide a specialised index to perform phrase searching that can reduce space significantly. The space taken by our index is comparable to the space taken by an Inverted Index with position lists(which it requires to perform phrase searches).

It is more profitable (by virtue of space alone) to use an FM Index(instead of CST) based index. While using an FM Index based approach we need to store the encoding of the tree as well. Space taken by the balanced paranthesis can be reduced considerably while using this for a phrase index. In addition to this space drop there is an advantage provided by the FM Index.

Variation of retrieval time with number of words in the phrase query

From 5.15 we can see that the performance of the inverted index decreases when the number of words in the query increases while our index comparatively performs optimally.

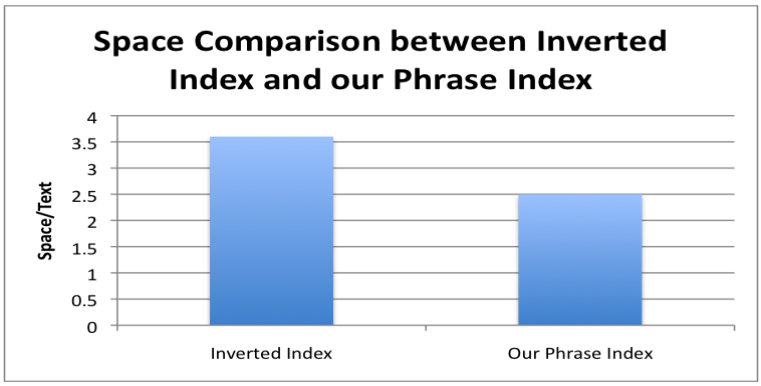


Figure 5.14: Space Comparison between Inverted Index and Our FM Index based index

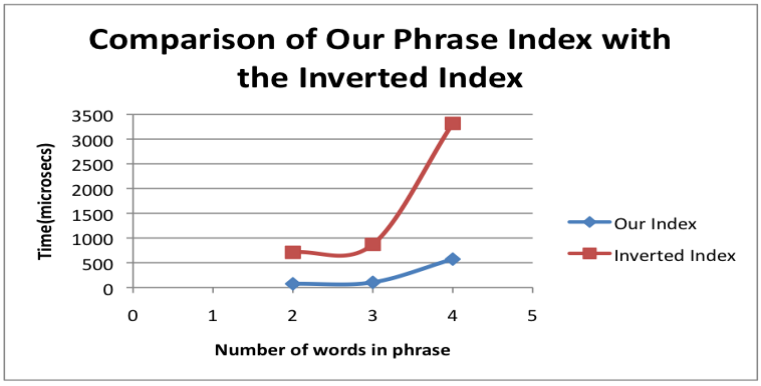


Figure 5.15: Shows the effect of increasing the number of words in query on retrieval time

Variation of retrieval time with input data size(for 2 word queries)

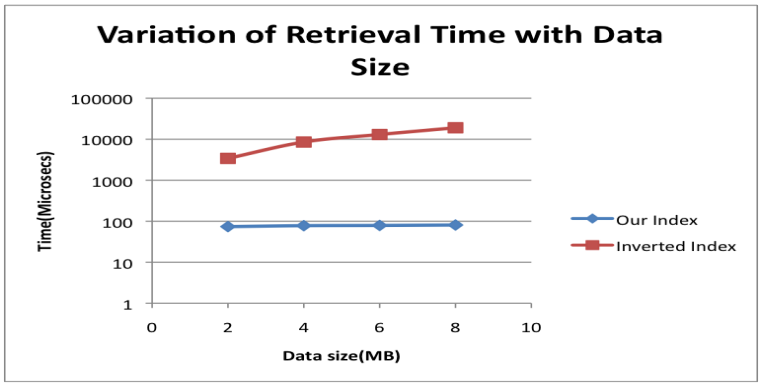


Figure 5.16: Shows the effect of increasing data size on retrieval time

From 5.16 we can see that the performance of the inverted index decreases with increase in data size. In figure 5.16 the retrieval time of our index cannot be clearly noted as the difference in retrieval time between the inverted index and our index is very high.

5.4.9 Experiment 9

We compare our index with other indexes that perform Top-K document retrieval in 5.17.

Index	Index size	Query time (micro sec)
Wu/Muthu	250 x input	50
Culpepper et al	2.5 x input	700
Ours	10 x input	70

Figure 5.17: Shows the comparison between our index and other available Top-K indexes

Chapter 6

Conclusion and Future Work

In this thesis, we provided a practical and efficient retrieval system for Top-K documents. We showed an index with optimal retrieval time with considerably lower space bounds than the previously known indexes with competative time bounds. We also showed the effectiveness of this index by extending its functionality to retrieve Top-K documents based on arbitrary score functions (like page rank) and the ability to alter the index to provide application specific indexes with more optimal bounds for both space and time. The index can also be used to retrieve Top-K documents when two patterns are given as query in optimal time. Overall this index can prove to be a very effective and multifunctional index.

Future work can include checking the feasibilty of retrieving Top-K document for two pattern queries using their tf-idf scores. The practical optimality of the compressed space index provided in Hon et al [14] remains to be seen.

Bibliography

[1] http://en.wikipedia.org/wiki/invertedindex.

[2] http://en.wikipedia.org/wiki/suffixarray.

[3] A. Andersson, N. Larsson, and K. Swanson. *Suffix Trees on Words ,Combinatorial Pattern Matching*, volume 1075/1996 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 1996.

[4] D. Arroyuelo, R. Canovas, G. Navarro, and K. Sadakane. Succint trees in practice. *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2010.

[5] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94. Springer-Verrlag,Lecture Notes in Computer Science, 2000.

[6] M. A. Bender, M. Farach-Colton, G.Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, pages 75–94, 2005.

[7] I. Bialynicka-Birula and R.Grossi. Rank-sensitive data structures. *In Proceedings of International Symposium on String Processing and Information Retrieval*, pages 79–90, 2005.

[8] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, 1994.

[9] D.E.Knuth, J.H.Morris, and V.B.Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, pages 323–350, 1977.

[10] P. Ferragina, M. G., M. V., and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[11] P. Ferragina and G. Manzini. Opportunistic data structures with applications. *Foundations of Computer Science*, 41:390, 2000.

[12] P. Ferragina and Venturini.R. The compressed permuterm index. *ACM sigir conference on Research and development in information retrieval*, pages 535–542, 2007.

[13] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. *Proceedings of Symposium on Theory of Computing*, pages 135–143, 1984.

[14] W.-K. Hon, R. Shah, and J. S. Vitter. Space -efficient framework for top-k string retrieval problems. *Foundations of Computer Science*, 2009.

[15] J.Fisher and V.Heun. A new succint representation of rmq-information and improvements in the enhanced suffix array. *In Proc. of Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 459–470, 2007.

[16] J.Zobel and A.Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2), 2006.

[17] K.Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix arrays. *Proceedings of the International Symposium on Algorithms and Computation*, pages 410–421, 2000.

[18] K.Sadakane. Space-efficient data structures for flexible text retrieval systems. *Journal of Discrere Algorithms, ISAAC*, pages 12–22, 2002.

[19] K.Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, pages 589–607, 2007.

[20] K.Sadakane. Succint data structures for flexible text retrieval systems. *J.Discrete Algorithms*, pages 12–22, 2007.

[21] Y. Matias, S.Muthukrishnan, S. Sahinalp, and J.Ziv. Augmenting suffix trees with applications. *In Proceedings of European Symposium on Algorithms*, pages 67–78, 1998.

[22] E. M.McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, pages 262–272, 1976.

[23] J. I. Munro and V. Raman. Succint representation of balanced parantheses, static trees and planar graphs. *SIAM Journal on Computing*, 31, 2001.

[24] U. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22:935–948, October 1993.

[25] N.Valimaki and V.Makinen. Space efficient algorithms for document retrieval. *In Proceedings of Symposium on Combinatorial Pattern Matching*, pages 205–215, 2007.

[26] O.Berkman and U.Vishkin. Recursive star-tree parallel data structure. *SIAM J.Computing*, pages 221–224, 1993.

[27] R.Grossi and J.S.Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *Pro ACM Symp. on Theory of Computing*, 32:397–406, 2000.

[28] B. Schieber and U.Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J.Computing*, pages 1253–1262, 1988.

[29] S.Muthukrishnan. Efficient algorithms for document retrieval problems. *Proceedings of Symposium on Discrete Algorithms*, pages 657–666, 2002.

[30] T.Kasai, G.Lee, H.Arimura, S.Arikawa, and K.Park. Linear-time longest-common-prefix computation of suffix arrays and its applications. *In Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, 2001.

[31] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

Vita

Sabrina Chandrasekaran was raised and did most of her schooling in Bangalore, India. She did her undergraduate degree in electrical and electronics engineering in the National Institute of Technology, Surathkal, Karnataka, India from the year 2004 to 2008. She then came to the United States for her master's degree with the Department of Computer Science at the Louisiana State University, Baton Rouge, Louisiana. She wrote her thesis under the able guidance of Dr. Rahul Shah.