EFFICIENT AND SCALABLE INDEXING TECHNIQUES FOR SEQUENCE DATA MANAGEMENT

ANAND THAMILDURAI

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL REQUIREMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

APRIL 2007

Your file Votre référence
ISBN: 978-0-494-34725-6
Our file Notre référence
ISBN: 978-0-494-34725-6

ABSTRACT

EFFICIENT AND SCALABLE INDEXING TECHNIQUES

FOR SEQUENCE DATA MANAGEMENT

ANAND THAMILDURAI

Sequence data is one of the rapidly growing types of data. New efficient and scalable techniques are needed to support fast access to this type of data. We study indexing techniques for sequence data, especially biological sequence data. The existing solutions for this type of data either support efficient index construction for long sequences or support fast search, but not both. We propose two new indexing techniques, Suffix Tree Top-Down 64 bit and Suffix Tree Depth-First 64 bit, which offer a tradeoff between scalable index construction, index size, and support of fast search. They differ in the order in which the index nodes are recorded but have similar performance. We compare our techniques with the best known existing techniques, which are based on suffix trees (TDD) or suffix arrays (ESA). The results of our extensive experiments show that while our proposed techniques have a slightly slower construction time for small sequences and larger index size compared to TDD, they outperform TDD in search. We further show that for very large sequences, such as the human genome (about 3GB), our techniques are superior to TDD due to the use of dynamic buffering and better index representation. Compared to the most search efficient in-memory indexing technique, ESA, our proposed techniques are slower in construction but have comparable index size and search performance. The main advantage of our techniques over ESA is that they are disk-based and can handle large sequences.

Acknowledgments

It's a great pleasure to thank the many people who made this thesis possible.

Foremost, I would like to gratefully acknowledge and thank my supervisor Dr. Nematollaah Shiri for his continuous support, motivation, and encouragement throughout my graduate studies. His knowledge, constant advices and many insightful conversations has been very invaluable to me. Special thanks to him for giving me opportunity to publish my works.

I would like to thank my colleague and friend Mihail Halachev for his support through out my thesis work. He has always been a constant source of knowledge, help and ideas whenever I needed them.

I also wish to thank all my colleagues in the database research labs for their technical ideas, informal discussions and support. Many thanks to all my friends and teachers who have always lent a helping hand during my difficult times.

Thanks to NSERC, Genome Quebec and the Dept. of Computer Science and Software Eng., Concordia University for their support.

Lastly, and most importantly, I am forever indebted to my parents, Thamildurai P. and Manonmani C. for their unconditional love, inspiration, endless patience and unwavering faith in me. To them I dedicate this thesis.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Advances in technology in the last decade has resulted in the generation of enormous amount of data that needs to be organized, stored, processed, and queried efficiently. Some of these datasets comply with the traditional database model (i.e. tuples with attributes). The existing query processing and optimization techniques in conventional databases are mainly suitable for such data. However, a substantial amount of data does not follow the relational database model. Thus, new scalable, efficient and versatile techniques that access and analyze these data are crucial to the success of emerging applications that use such data.

One such type of fast growing data that does not comply with the traditional relational database model is sequence data. Sequence data is ubiquitous and includes textual data in the web, digital libraries, DNA (genome) databases, protein databases, etc. Sequence data is represented as strings over some alphabet of finite size. For example, DNA sequences are represented as strings over the nucleotide alphabet $\Sigma = \{A,C,G,T\}$, and proteins which are the building blocks of life are represented as strings over the amino acid alphabet of size 23. Many of these datasets, especially the genomic datasets have been growing at an exponential rate over the past decade. For example, the most popular and widely used biological dataset, GenBank along with its collaborating DNA and protein

databases with data encoded as long strings is more than 100 Gigabase pairs (Gbp) in size (1 bp = 1 character). The size of GenBank alone doubles every 17 months. The growth of GenBank from 1982-2007 [NCBI, 2007] is shown in Figure 1. Trace Archive is another example dataset that stores all the sequence data produced and published and it doubles every 10 months. The size of this dataset is more than 22 terabytes (22,000 GB) and is one of the biggest datasets in the world [Sanger, 2006].

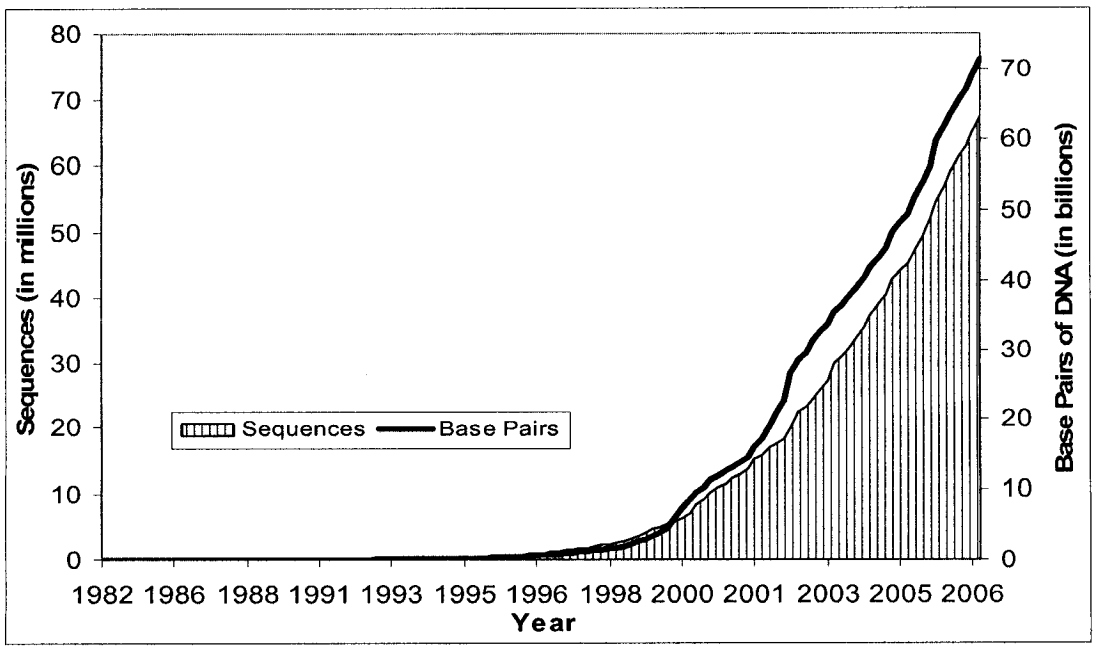


Figure 1 Growth of GenBank (1982-2007)

1.1 Motivation

The exploding growth of sequence data, especially publicly available biological data, poses new challenges to existing database storage and querying techniques. Unlike natural texts which can have words, paragraphs, etc, an important characteristic of biological sequence data is that they do not have any structure. As a result, traditional database techniques are inadequate for efficient storage and retrieval of such data. For the

management of such data, development of new techniques is thus necessary. Basically, we need to create an index for this data, and reduce the number of disk access required for processing queries. An important question in this context is: What is an appropriate model for indexing sequences? Two main issues that need to be addressed are the required storage space for the index and its construction time. Further, in terms of query processing, searching in molecular sequence data is central to computational molecular biology, and hence a desired data model should minimize the number of disk I/O operations in order to provide fast access and efficient support for various types of search applications. Examples of such applications include local and global alignment, searching for motifs, sequence-tagged sites (STSs), expressed-sequence-tags (ESTs), finding exact and approximate repeats and palindromes, recognizing DNA contamination, DNA sequence assembly, etc. Given a query pattern, a core part of the solution in these applications is to find substrings in the dataset which are the same or similar to the query pattern.

Thus, a desired technique should satisfy the following:

1) Handle enormous amount of data efficiently.

2) Support a wide variety of searches on sequence data.

3) Scale well with increasing size of the sequences.

Our objective in this thesis is to provide such a solution technique that can be used on standard desktop computers. The focus of this thesis is on the data model and storage structure algorithms. We concentrate more on suffix tree based indexing techniques. This is because suffix trees are versatile data structures that efficiently support more than 20 search applications [Gusfield, 1997]. In this context, we propose a new indexing

technique that can handle large sequence data. We also evaluate and compare the proposed techniques with the existing techniques based on the index construction time, index size, and search efficiency.

1.2 Contributions

This work is a part of an ongoing project in our database research lab on the management of large sequence data. My main contribution in this work is development of suffix tree based indexing techniques. In particular, my research focus was on developing an efficient disk-based suffix tree construction algorithm, crucial to the scalability of the indexing technique. This work builds on our earlier result which was based on a 32 bit representation model of the index [Halachev et al., 2005], which was further extended to a 64 bit model recently [Halachev et al., 2007].

Details of the main contributions of this thesis are as follows:

1. The development and implementation of Suffix Tree Top-Down 32 bit (STTD32) and Suffix Tree Depth-First 32 bit (STDF32) [Halachev et al., 2005], two techniques that extend the most space efficient suffix tree based technique, WOTD [Giegerich et al., 2003] is one of the contributions of this thesis. STTD32 is a disk-based extension of WOTD while STDF32 differs from WOTD in the order in which the nodes are evaluated and recorded.
2. We then compare STTD32 and STDF32 to the most efficient and scalable suffix tree based technique, Top Down Disk-Based (TDD) [Tata et al., 2004]. We experimentally show that our STTD32 and STDF32 outperform TDD in terms of

construction time, storage space and exact match search performance. Compared to STTD32, STDF32 has similar construction time and same storage space but has a better locality of reference for construction and search compared to STTD32 [Halachev et al., 2005]. We also discuss the various properties, advantages and limitations of our techniques.

3. In order to overcome the limitation of the 32 bit techniques, we propose two new techniques, Suffix Tree Top-Down 64 bit (STTD64) and Suffix Tree Depth-First 64 bit (STDF64) [Halachev et al., 2007].
4. We experimentally show that the 64 bit techniques outperform the 32 bit techniques and TDD in terms of search time, however this is achieved at the cost of increased construction time and storage cost. We also demonstrate that our proposed representations have similar search performance compared to the most efficient memory-based solution, Enhanced Suffix Array (ESA), for the sequences they can handle. While ESA can support only sequences of size up to 400 MB (theoretical limit), our 64bit techniques can support sequences of up to 4 GB. We finally provide an overview of the various techniques that are suitable for different size of datasets.

1.3 Thesis Outline

In the next chapter, we present the background and related work. In related work, we review existing indexing techniques for management of biological sequences, and

evaluate them based on scalability and efficiency of construction, space requirements, and search algorithm.

In Chapter 3, we introduce two existing indexing techniques which inspired our work, WOTD technique [Giegerich et al., 2003] and TDD technique [Tata et al., 2004].

In Chapter 4 we discuss the preliminary study we conducted through the comparison of TDD technique and two extensions of WOTD technique, STTD32 and STDF32 based on their construction and search performance.

In Chapter 5, we propose our new 64 bit indexing techniques, and study their performance compared to STTD32, STDF32, TDD, and ESA.

We then conclude with a summary of our work and present a list of possible future works in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we first define sequence data, their applications, and techniques used to improve access to such data. Next, we provide a comprehensive study of the various existing indexing techniques for sequence data. We conclude this chapter with a discussion on the various index structures highlighting their strengths and weaknesses on the basis of their index construction time, index size, and search performance.

2.1 Sequence Data

A sequence D is defined as a string of symbols over some alphabet of finite size. For example, the sequence D = THESIS is a sequence over the 26 English alphabets and the number of characters in the sequence, $|D| = 6$. For any sequence D, $D[i..j]$ is a contiguous substring of D that starts at position i and ends at position j. Note that $0 \leq i \leq j < |D|$. If position $i = 0$, then $D[0..j]$ is the prefix of D that ends at position j. If $j = |D|-1$, then $D[i..j]$ is the suffix of the sequence D that begins at position i. In the above example "*THE*" is a prefix of D that ends at position $j = 2$ while "*SIS*" is a suffix that starts at position $i = 3$. A sequence database includes finitely many sequences.

Note that the terms "text", "string" and "word" are used interchangeably in computer science literature to represent sequences. We will use the term "sequences" in this thesis. Even though the terms "sequence" and "string" are synonymous in literature, "subsequence" and "substring" represent different objects in biological literature. Substrings must occur contiguously in D, while characters in subsequences might be interspersed with characters not in the subsequence. In biological literature the terms "sequence" and "subsequence" are also used synonymously [Gusfield, 1997]. In this thesis, we distinguish between a sequence and subsequence and also between substring and subsequence.

Examples of sequence data include digital libraries, DNA (genome) databases, protein databases, financial time series, web access patterns, data streams generated by micro sensors, etc. In our work, we consider two types of sequence data: 1) biological sequence data - DNA and protein sequence data, and 2) natural text.

Biological Sequence Data

Biological sequence data is the main reason why sequence data has received much attention in recent times. Analysis of such data assists the biologist in understanding and explaining a number of biological phenomena from the structure of the biomolecules and their interaction to the behavior of living things. This knowledge facilitates in discovering the history of life, fighting diseases and also developing new drugs. There are already huge amount of public and commercial biological datasets and they are growing in size at an enormous rate. Some of the popular biological datasets are GenBank [GenBank, 2005], European Molecular Biology Laboratory (EMBL) Nucleotide Sequence Database

(EMBL-Bank) [EMBL, 2005], DNA Database of Japan [DDBJ, 2005], and Swiss-Prot protein database [SProt, 2005] to name a few. Biological data is mainly in the form of DNA sequences of nucleotides or amino acid sequences of proteins. They also include other data like functional and structural information, which we do not consider in our study.

Deoxyribonucleic acid (DNA) is a nucleic acid that contains the genetic instructions for the development and functioning of living organisms. All living creatures contain DNA genomes. DNA is a long polymer made from repeating units called nucleotides. These repeating units are very small; however, DNA polymers can be enormous molecules containing millions of such nucleotide units. For example, the length of the largest human chromosome, Chromosome 1, is around 247 million base pairs (bp), where 1bp is equal to 1 symbol. The total length of all the 24 human chromosomes is nearly 3 billion bp.

Formally, a DNA sequence is considered as a sequence over the nucleotide alphabet $\Sigma = \{A, C, G, T\}$. Each of these symbols represents an actual nucleotide unit. DNA sequences are usually stored in FASTA format in sequence databases. Figure 2 shows an example DNA sequence taken from the NCBI website [NCBI, 2007].

```
>gi|125655869|emb|CS470463.1| Sequence 54 from Patent EP1748083
TGACTACTTTTGACTTCAGCCAGTATATGAAATTGGATATTGCAGCAGTCAGAGCCCTTAACCTTTTTCA
GGTAAAAAAAAAAAAAAAAAAAAAGGGTTAAAAATGTTGATTGGTTAA
```

Figure 2 DNA Sequence in FASTA format

The first line in the FASTA format is the header information for the sequence. It contains a unique identifier for the sequence in the database, its name and other information. In the subsequent lines the actual sequence is represented. Each of these lines is at most 70 symbols in length.

Proteins are linear polymers built from different amino acids. Proteins are assembled from amino acids using information encoded in genes. They are essential parts of all living organisms and participate in every process with the human cell. They also play an important role in immune responses, cell cycle, diet, etc.

For algorithmic purposes a protein is considered a sequence over an alphabet of 23 amino acids. Similar to DNA sequences, protein sequences are commonly stored in FASTA format. An example of a protein sequence taken from NCBI website [NCBI, 2007] is shown in Figure 3.

```
>gi|267936|gb|AAA00327.1| Sequence 2 from Patent US 4452775
AFPAMSLSGLFANAVLRAQHLHQLAADTFKEFERTYIPEGQRYSIQNTQVAFCFSETIPAPTGKNEAQQK
SDLELLRISLLLIQSWLGPLQFLSRVFTNSLVFGTSDRVYEKLKDLEEGILALMRELEDGTPRAGQILKQ
TYDKFDTNMRSDDALLKNYGLLSCFRKDLHKTETYLRVMKCRRFGEASCAR
```

Figure 3 Protein Sequence in FASTA format

Applications that operate on biological data typically perform one or more of the following types of operations as a part of more involved tasks - exact match search, similarity searching (*k*-mismatch and *k*-difference), finding repeats, etc. For example BLAST (Basic Local Alignment Search Tool) [Altschul et al., 1990] [Altschul et al., 1997], a popular search tool among biologists, performs exact match as a sub-task of the alignment process.

An important characteristic of biological sequence data is that it cannot be broken into separate meaningful words/substrings. As a result most of the traditional indexing techniques for sequence data perform badly for this type of data. Further, the complexity, heterogeneity and quantity of biological data also pose new challenges. Therefore, new efficient and scalable techniques are required to be designed to store, access and manipulate these data. This is the main motivation of this thesis work.

Natural Text

Natural text data consists of textual information stored in files. Textual information stored in digital libraries, like project Gutenberg [Guten, 2005], constitute to this type of data. Indexing of natural text is mainly performed to support full text searching. Many traditional indexing techniques work for this types of data. However, the recent explosive growth of these kinds of data, especially in the World Wide Web, poses a challenge to researchers. We mainly use this type of data in our experiments to show the effect of increased alphabet size on the performance of the index we propose.

Common search applications using natural text include exact match search or wild card search on the dataset. Some search applications apply natural language concepts to the query pattern before the actual search. This is done to predict possible synonyms and also to remove common words like "it", "this", "the", etc.

2.2 Improving Access to Sequence Data

The naïve method of searching in sequence data performs a sequential traversal of the entire data. This method of searching sequence data is slow, especially when the sequence size is large, due to the fact that the entire dataset has to be scanned. In order to improve the access to such large sequences a preprocessing step may be useful. There are two types of preprocessing possible: (1) preprocessing the query pattern P, and (2) preprocessing the sequence dataset D.

2.2.1 Preprocessing the Query Pattern

In this approach, the query pattern provided by the user is preprocessed and useful information is stored in temporary structures. The information is then used to reduce the work done while searching. An advantage of this method is that since the query pattern is usually a small fraction of the size of the whole database, preprocessing is fast. The major disadvantage is that the cost of preprocessing is incurred by the user every time the database is queried.

An example tool that preprocesses the query is BLAST. It is a search tool which performs alignment of two or more sequences. For this purpose, the given query pattern P is broken into a number of disjoint substrings called words of some predefined length l. Then, the database sequences are traversed sequentially, and exact matches of the query words are found in the database. These exact matches are then used as seeds for the alignment process. The sequential traversal of the database results in slower search efficiency. For example, a BLAST search for 3 human chromosomes of total size 294Mbp (~294MB, i.e., 10% of the human genome) was able to answer 99 queries (of length between 429 and 5999 bp) in around 62 hours, using an Enterprise 450 SUN computer with 2GB RAM [Hunt et al., 2001].

2.2.2 Preprocessing the Data

A large part of the sequence data is mostly stable, i.e., they are not changed too often. Unlike a query pattern, which is known only at the time of searching, the sequences are known beforehand and the cost of preprocessing the sequence can be amortized over

many queries. Hence, it is beneficial to preprocess the sequence to improve search. Creating an index on the sequence data D is an example of this kind of preprocessing. Once an index is built, the user queries are answered by performing more focused accesses to D, using the information stored in the index. In this technique, the user does not incur any cost of preprocessing. Several biological tools developed follow this idea of preprocessing of data by building and using indexes. Examples include MUMmer [Delcher et. al., 1999] and REPuter [Kurtz and Schleiermacher, 1999].

In this work, we use preprocessing of data to provide effective search in sequence datasets. In the next section, we review major indexing techniques developed for sequence data.

2.3 Indexing Techniques for Sequence Data

There are a number of techniques for indexing sequence data. They include techniques based on Inverted files [Witten et al., 1999], q-grams [Navarro and Baeza-Yates, 1998], Tries [Fredkin et al., 1960], PATRICIA tries [Morrison, 1968], SPINE [Neelapala et al., 2004], suffix trees [Weiner, 1973], suffix arrays [Manber and Myers, 1991] and string B-Tree [Ferragina and Grossi, 1999]. A brief discussion of these index structures is presented next.

2.3.1 Inverted Files

An inverted file [Witten et al., 1999], also referred to as an inverted index, is an index structure which maps words to a sequence of words/documents. It is a type of secondary

index in which instead of creating a separate index for each attribute (i.e. for each word), the indexes are combined. Indirect buckets are used for space efficiency. While inverted files are useful for natural texts, a disadvantage of inverted file is that it cannot be used for biological sequence data, because of their inherent lack of structure [Hunt et al., 2001].

2.3.2 *q*-grams

The q-gram data structure was proposed by [Navarro and Baeza-Yates, 1998] as a sequence indexing technique for approximate pattern matching, which is one of the most common search operations performed on biological sequence data. Sequences of a fixed length q appearing in the original input sequence D are stored in the index, along with a pointer to their exact location in the sequence. The input query pattern P is cut into small pieces of same length q, which are then queried against the index. All the pieces of the query pattern of a fixed length are processed similarly to find all possible candidate positions and the union of the results is taken. Based on the experimental results, the value of q is between 3 and 5 as tradeoff between the index size and search time. There is also some loss in information because all query pattern pieces of length more than q are truncated. The proposed index construction algorithm for q-grams has linear time complexity.

[Miller et al., 1999] and [Burkhardt et al., 1999] have implemented q-grams as indexes for their search applications. [Miller et al., 1999] uses q-grams to identify vector contamination in EMBL databases [EMBL, 2005] while [Burkhardt et al., 1999] use q-grams along with suffix arrays to search biological sequence databases. While these

approaches based on q-grams are fast, they do not perform well for searches with low similarity, which are significant from a biological point of view [Navarro, 2000].

2.3.3 Trie Based Techniques

The *trie* (from the word re*trie*val) data structure was introduced by [Fredkin et al., 1960] and is used for indexing sets of key values of varying sizes. The trie data structure is a multi-way ordered tree data structure used for storing and retrieving information. It has been used successfully on sequences that can be broken into distinct words like dictionaries and also supports biological sequences. The idea of a trie is that all sequences that share the common prefix share a common node. Any node in a trie has at most n+1 children, where n is the number of symbols in the alphabet. The one extra child is for the unique terminal symbol. The terminal symbol is used to ensure that no two suffixes have the same prefix; otherwise there is a possibility that the same node can be both a branch and leaf. In our work we use $ as the terminal symbol in a sequence. For example, if we construct a trie for a set of sequences in the English language, then every node would have at most 27 children. Figure 4 shows a trie data structure for the following 5 sequences – DIG, DIM, DINNING, GOD, GOOD.

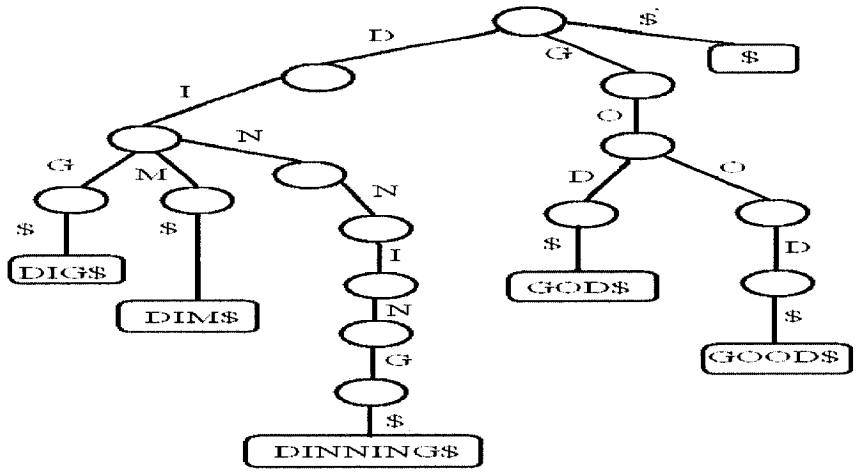


Figure 4 (Non Compact) Trie Data Structure

In the figure, branch nodes are marked by circles and leaf nodes by curved rectangles. Once a sequence has been indexed, any key pattern P of size m in the sequence can be found in $O(m)$ time. One of the main disadvantages of using this index is its space utilization, since every symbol in the given input is represented as an edge in the trie. The trie shown in Figure 4 is called a "Non-Compact" trie, since every edge represents a symbol from the alphabet set.

A variation of tries called a "Compact Trie" was proposed to reduce the space requirements of a non-compact trie by trimming the chains that lead to the leaves. An example of a compact trie is shown in Figure 5.

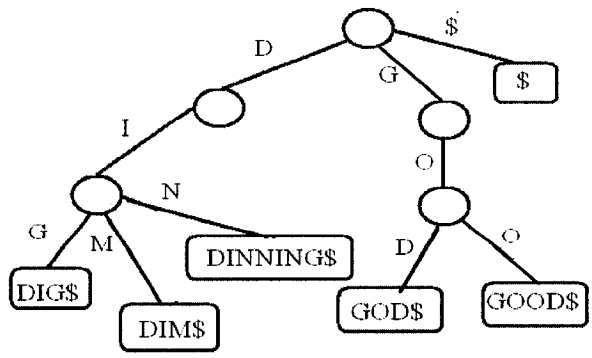


Figure 5 Compact Trie

To reduce the space requirement of these data structures further, the PATRICIA data structure was proposed. PATRICIA stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [Morrison, 1968]. A PATRICIA trie is a compact trie where each edge can represent more than one symbol. Thus in a PATRICIA trie all unary nodes are compacted. A PATRICIA trie for a sequence of length n contains n-1 branch nodes and n leaf nodes. The PATRICIA trie has the same search complexity as the trie, but reduces the space requirement considerably. Figure 6 shows the PATRICIA trie for the sequences represented by the trie in Figure 4.

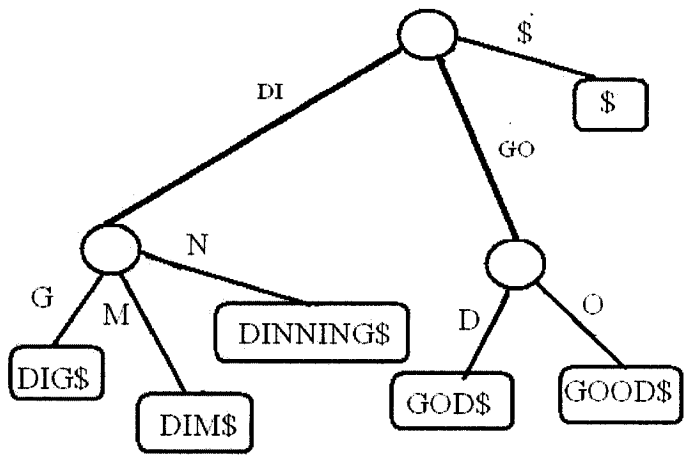
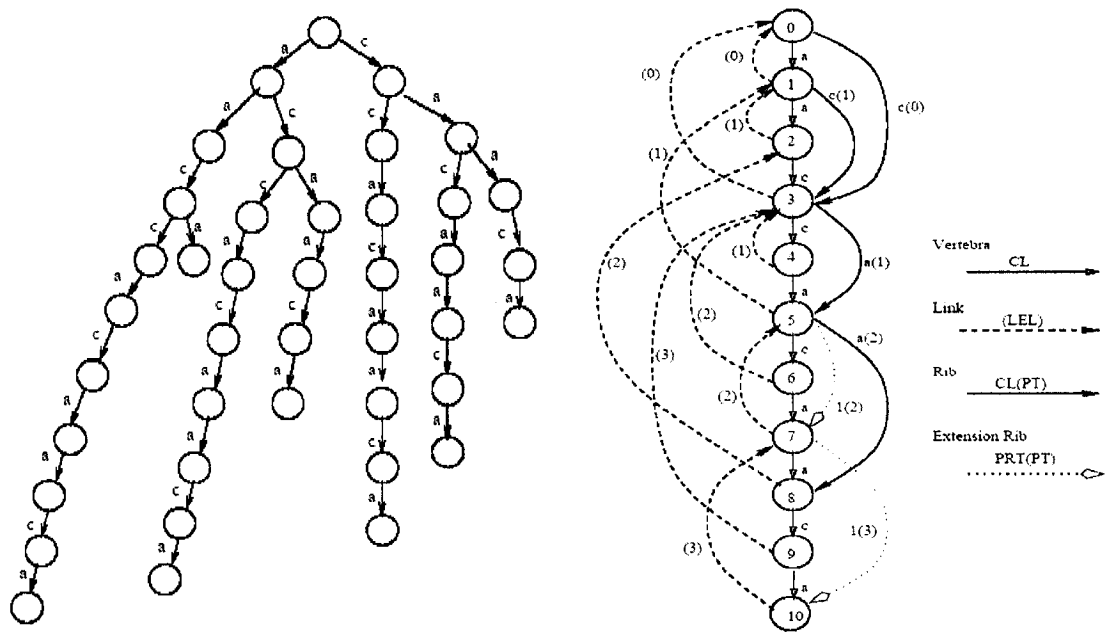


Figure 6 PATRICIA Trie

Another alternative to the vertical compaction of the trie as done in Compact Tries and PATRCIA trie is to compact all related nodes horizontally. This approach has been used in the String Processing INdexing Engine (SPINE) [Neelapala et al., 2004]. The SPINE data structure was proposed as an alternative to the suffix trees (discussed next) and is obtained by the horizontal compaction of the trie. While SPINE offers all the standard functionalities of the suffix tree, the number of nodes in the SPINE data structure is always equal to the sequence length (in suffix trees this number might be as large as twice the sequence size).

The trie and its corresponding SPINE data structure for the sequence D = aaccacaaca, taken from [Neelapala et al., 2004] are shown in Figure 7.



(a) (b)

Figure 7 (a) Trie for D = aaccacaaca and (b) corresponding SPINE

In the SPINE data structure, shown in the figure, vertebras are forward directed edges where each corresponds to a character in the input string. This character provides the character label (CL) for the vertebra. They appear in the same order in which they appear in the string. For example, in the above figure the path traced by the vertebra starting from the root node 0 and ending at node 10, spells the input string, "*aaccacaaca*". This forms the backbone of all the nodes. In addition there are edges called ribs and extension ribs that are downstream edges. Each rib is also labeled with a CL corresponding to the character it represents in the associated suffix. The set of forward edges collectively represent all possible suffixes of the data string, and this information is used during the search process. For example the forward edges starting from node 1 and ending at node 10 encode the suffix "*accacaaca*". Horizontal merging of the trie data structure,

represented as a SPINE might result in false positives, i.e., invalid substrings may arise. To avoid this, the SPINE encodes two more integer data – Pathlength Threshold (PT) and Longest Early-Terminating Suffix Length (LEL).

The construction process starts with the root node and then for each new character in the sequence, a node is appended to the tail. The associated links are then created as required [Neelapala et al., 2004]. The actual characters from the sequences are stored in the index structure and hence once the SPINE is created, the sequence is no longer needed. This is in contrast with suffix trees and suffix arrays (discussed next) where the sequence is needed even after the index construction.

The space requirement for SPINE is 48.25 bytes per node in the worst case and is around 12 bytes per node in an optimized version [Neelapala et al., 2004]. This optimization has been performed by reducing the space required for certain information associated with each node and this is determined based on their experimental results. The largest sequence they consider is human chromosome 19 (57.5 Mbp). These optimizations may not work for sequences as large as the human genome (3 billion bp). Further, the disk based performance is not comparable to the most efficient suffix tree and suffix array construction algorithms. The reported SPINE disk construction times for human chromosome 21 (which is of size 28.5 million bp) on a Pentium 4 machine @ 2.4GHz, with 1GB RAM is around 9 hours [Neelapala et al., 2004].

2.3.4 Suffix Trees

A suffix tree is a suffix trie in which every unary node is compacted and collapsed. Suffix tries are a type of trie where indices corresponding to each symbol in the sequence are

used instead of the actual symbol. A suffix tree T for an n-character sequence D (which includes the terminal character \$) is a rooted directed tree with exactly n leaves numbered 1 to n. Each branch node, other than the root, has at least two children and each edge is labeled with a nonempty substring of D. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of D that starts at position i, i.e., it spells out $D[i...n]$ [Gusfield, 1997]. Once a suffix tree is constructed, it allows all z occurrences of a query pattern P to be located in $O(|P| + z)$ time, independent of the size of the sequence D. The suffix tree data structure is versatile and supports as much as 20 sequence processing applications [Gusfield, 1997]. Due to this versatility exhibited by suffix trees, they have been implemented in some bioinformatics applications. These include MUMmer [Delcher et al., 1999], and REPuter [Kurtz and Schleiermacher, 1999]. MUMmer is used for aligning entire genomes, while REPuter builds a suffix tree on the fly and computes the repeats.

The suffix tree for our example sequence D = AGAGAGCTT\$ is shown in Figure 8. In the figure, the numbers below each leaf in the tree represent the starting positions of the suffix that is encoded by the path starting at the root and ending at that leaf. For example the leaf numbered 0 encodes the suffix AGAGAGCTT\$, i.e., the actual sequence D. The leaf number 6 spells the suffix CTT\$ which starts at position $D[6]$.

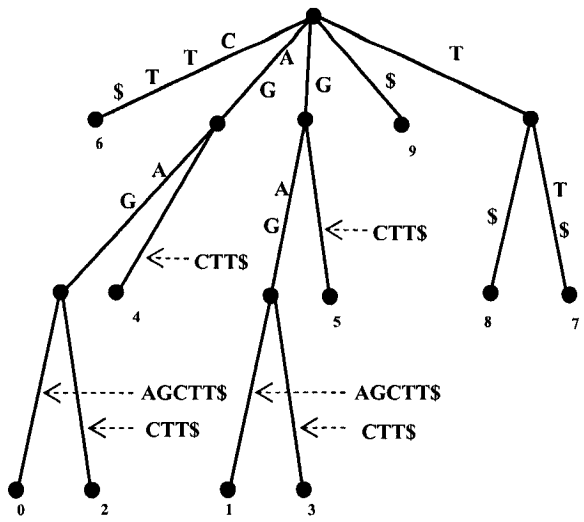


Figure 8 Suffix Tree

A number of suffix tree representations along with their construction algorithms have been proposed in the literature. The first linear-time algorithm for constructing suffix trees was proposed by [Weiner, 1973]. [McCreight, 1976] proposed a space efficient algorithm to build suffix trees in linear-time. [Andersson and Nilsson, 1995] proposed a LC-trie representation of the suffix tree while [Kurtz, 1999] proposed two alternative suffix tree representations, one based on linked list representation and the other based on hash tables.

The most popular linear-time construction algorithm was proposed by [Ukkonen, 1995]. It is conceptually different from [Weiner, 1973] and [McCreight, 1976], and its major advantage is its simplicity of implementation. All these algorithms are memory-based and require $O(|D|)$ time and space. One of the main implementation requirements for linear time suffix tree construction algorithms is the use of suffix links [Gusfield, 1997]. A suffix link is a pointer from one internal node to another. While suffix links enable linear time construction of suffix trees, it has been shown in [Hunt et al., 2001] that their

existence results in random accesses to the tree during the construction of the suffix tree. As a result algorithms that implement suffix links are not scalable.

Due to the versatile applications of suffix trees, research efforts have been made to develop disk-based construction algorithms for small sequences. [Bieganski, 1995] built suffix trees for sequences of size up to 1Mbp, [Navarro and Baeza-Yates, 2000] also implemented a suffix tree construction algorithm for similar sequence sizes. From their experiments [Navarro and Baeza-Yates, 2000] concluded that suffix trees in excess of RAM size cannot be built. [Kurtz, 1999] estimated that the suffix tree for sequences as large as the human genome (3GB) would require 45.31 gigabytes of main memory on a 64 bit architecture machine. In [Hunt et al., 2000] suffix trees for sequences as large as 20.5 Mbp were built on an Enterprise 450 Sun machine, with four 400 MHz UltraSPARC-II CPUs and 2GB RAM. The constructed index was around $21n$, in size.

While suffix trees support numerous applications [Gusfield, 1997] and have wide acceptance by theoretical computer scientists, their use in sequence applications, has not been widespread compared to other techniques like finite automata or hashing techniques. This is because the construction of suffix trees for very large sequences was not possible. [Skiena, 1998] has observed that suffix tree data structures are the data structure with the highest need for better implementations. This is due to the following reasons [Giegerich et al., 2003]:

1) The linear-time construction algorithms proposed by [Weiner, 1973], [McCreight, 1976], and [Ukkonen, 1995] are difficult to implement.
2) Although linear-time algorithms are available for construction, their poor locality of memory reference [Giegerich and Kurtz, 1995] and their memory bottleneck

due to the random access of the sequence being indexed [Farach et al., 1998] results in poor performance in cached architecture machines.

3) Suffix trees require an order of magnitude more space than the sequence itself. For example, [McCreight, 1976] requires around 28 bytes per character in the worst case. This combined with the fact that the linear-time construction algorithms have poor locality of memory references, results in fast degradation of performance when the sequence size increases.
4) As a result of all the above observations for most applications, the performance using suffix trees was inferior compared to other methods.

Much recent work has focused on addressing these limitations of suffix trees and making them a practical alternative for indexing large sequence data. [Hunt et al., 2001] identified that the main reason for this poor scalability of most suffix tree construction algorithms is due to the presence of suffix links. The presence of suffix links results in random access of the tree during the construction process. In this context, they proposed a new suffix tree construction algorithm that abandoned the use of suffix links and performs multiple passes over the sequence, constructing the suffix tree for a sub range of suffixes at each pass [Hunt et al., 2001]. The later was achieved by partitioning the sequence based on the first prefix length characters and constructing the suffix tree for each partition at a time. Once a suffix tree for a partition has been built they can then be moved to the disk and the processing of other partitions would continue. Using the proposed construction algorithm of complexity $O(n\log n)$ they were able to construct the suffix tree for a sequence of size 263 Mbp in 19 hours on a Solaris 7 Enterprise 450 SUN computer with 2GB RAM.

In a separate work aimed at reducing the size of the suffix tree, [Giegerich et al., 2003] proposed a new suffix tree representation and its corresponding construction algorithm. This new representation, which we shall refer to as Write Only Top-Down (WOTD) requires around 12 bytes per input character in the worst case and 8.5 bytes per input character on an average. This is the most space efficient suffix tree representation proposed in the literature. The time complexity of the corresponding construction algorithm is $O(n^2)$. Even though the construction algorithm has a quadratic time complexity, it exhibits good locality behavior and hence showed linear time complexity for moderate size sequences.

In a more recent work, [Tata et al., 2004] extended the work done by [Giegerich et al., 2003] and incorporated the partitioning ideas in [Hunt et al., 2001]. The proposed technique called Top-Down Disk-Based (TDD) technique is disk-based and incorporates partitioning of sequence and exploits the locality exhibited by WOTD, by incorporating buffering strategies. The resulting construction algorithm called Partition and Write Only Top-Down (PWOTD) is $O(n^2)$, but outperforms the Ukkonen's memory-based linear time algorithm even for short sequences. TDD has also shown to be superior to other alternative solutions based on suffix trees and suffix arrays like [Bedathur and Harista, 2004] and [Dementiev et al., 2005]. Using this algorithm, for the first time, the suffix tree for the entire human genome was constructed in about 30 hours [Tata et al., 2004].

Suffix trees can also be constructed for more than one sequence. These kinds of suffix trees are called Generalized Suffix Trees (GST) and were introduced in [Bieganski et al., 1994]. The corresponding construction and suffix tree traversal and matching algorithms

are also presented in [Bieganski et al., 1994]. The generalized suffix tree can be constructed in time proportional to the sum of all the sequence lengths.

2.3.5 Suffix Arrays

The size of suffix tree for a sequence is an order of magnitude greater than the sequence itself, and hence a suffix tree, is not useful when disk space is scarce. In order to overcome this, suffix arrays were proposed in [Manber and Myers, 1991] as a space efficient alternative to suffix trees. Given an n-character sequence D, a basic suffix array for D is an array of the integers in the range 0 to $n-1$, specifying the lexicographic order of the n suffixes of the sequence D [Gusfield, 1997]. A suffix array for the sequence D = AGAGAGCTT$ is shown in Figure 9.

0	1	2	3	4	5	6	7	8	9
A	G	A	G	A	G	C	T	T	$

0	AGAGAGCTT$
2	AGAGCTT$
4	AGCTT$
1	GAGAGCTT$
3	GAGCTT$
5	GCTT$
6	CTT$
7	TT$
8	T$
9	$

Figure 9 Suffix Array

The *basic* suffix array holds only integer values corresponding to the lexicographic position of the suffix in the sequence and so it requires less space compared to the suffix tree. It requires $4n$ bytes of space for the array (on 32 bit computers). Once the *basic* suffix array is constructed, any pattern P of length m in the suffix array can be found

using a binary search. As a result, the time complexity of an exact match search using a basic suffix array is O($m$log$n$). In order to improve the search efficiency for the exact match search problem an additional data structure which stores the longest common prefix (LCP) values was proposed [Manber and Myers, 1991]. The resulting suffix array, called augmented suffix array (asa), requires an additional $4n$ space, but still does not match the search efficiency of a suffix tree based approach.

The *basic* suffix array for any sequence D can be constructed in linear time by first constructing the suffix tree for the corresponding sequence [Gusfield, 1997]. In some recent work, [Kärkkäinen and Sanders, 2003], [Kim et al., 2003], and [Ko and Aluru, 2003], it has been shown that a direct linear time construction of the suffix array is also possible. Until recently, the use of *basic* suffix arrays as an alternative for suffix tree data structure was limited. This is because compared to the suffix tree the *basic* suffix array by itself cannot support a wide variety of search applications efficiently. Additional data structures and disk-based construction algorithms were therefore needed to improve the efficiency of suffix arrays.

In order to improve the search efficiency of *basic* suffix arrays, [Abouelhoda et al., 2002] proposed an *Enhanced* Suffix Array (ESA). The ESA is a collection of ten tables including the *basic* suffix array. They further show that every algorithm that uses the suffix tree data structure can systematically be replaced with an algorithm that uses an ESA [Abouelhoda et al., 2004]. The proposed ESA based algorithms have the same time complexity as the suffix tree based techniques. The improved search efficiency of ESA is obtained at the cost of increased storage space of around $13n$ bytes. A limitation of the ESA construction and search algorithms is that they are memory-based. Theoretically the

ESA can support indexing sequences of up to 400 million symbols on 32 bit architecture machines, provided there is enough memory available ([Vmatch, 2006]). A 64 bit version of ESA is also available, however, it is also bound to the size of the main memory. The ESA has been implemented in a commercial tool called Vmatch [Vmatch, 2006], the executable code of which was made available for our comparative performance evaluations.

Some studies on *basic* suffix array construction, [Itoh and Tanaka, 1999], [Seward, 2000], [Burkhardt and Kärkkäinen, 2003], and [Manzini and Ferragina, 2004] aim at using minimal working space. These in-memory algorithms have $O(n^2 \log n)$ worst-case time complexity (except [Burkhardt and Kärkkäinen, 2003], which is $O(n \log n)$) and are referred to as "*lightweight algorithms*", and require $5n$-$7n$ working space. Considering a typical RAM size of 2 to 4 GB available today for typical desktop computers, the upper bound on the size of the indexed sequences for these algorithms is around 400-500 million symbols, which is also the limit for the ESA technique. For a more detailed study of the popular in-memory *basic* suffix array construction algorithms, the interested reader may refer to [Puglisi et al., 2005].

For indexing long sequences, there are several proposed disk-based suffix array construction algorithms. Most disk-based *basic* suffix array constructions are inefficient and do not scale well for very long sequences. [Crauser and Ferragina, 2002] compare six disk-based suffix array construction algorithms. The size of the sequences used in this study ranges from 26 million symbols (for DNA) to 50 million symbols (for random data). In a recent work, [Dementiev et al., 2005] has noted that the techniques in [Crauser et al., 2002] are bounded to a 2GB external memory.

The most efficient disk-based suffix array construction algorithm was proposed in [Dementiev et al., 2005]. The proposed algorithm, called DC3, is pipelined, I/O optimal and constructs only the *basic* suffix array. They further compare the performance of DC3 with other suffix array construction algorithms they implemented and show its superiority experimentally. [Dementiev et al., 2005] were able to construct the *basic* suffix array for the entire human genome in about 10 hours, using a computer with two 2 GHz processors, 1GB RAM, and four 80GB hard disks. Very recently, [Kulla and Sanders, 2006] have proposed pDC3, a parallelized version of the DC3 algorithm, using which they were able to construct the *basic* suffix array of the entire human genome in 90 seconds. However this requires enormous computing power. They used a system with 64 dual processor nodes using Itanium 2 processors with 1.5 GHz and 6MB cache. The machine had 64 x 12 GB of main memory. While this result is important, indicating "parallelizability" of the index construction algorithm for sequences, in this work, we are interested in techniques that can be used on typical desktop computer.

It should be noted that two memory-based suffix array construction techniques have also succeeded in creating the *basic* suffix array for the human genome. This was possible either using a computer with huge main memory or a supercomputer. [Sadakane and Shibuya, 2001] constructed the compressed suffix array for the entire human genome in around 7 hours using a super computer with 64 GB of memory. In [Lam et al., 2002] the compressed suffix array for the human genome was constructed in 21 hours using a computer with 3 GB RAM. Both construction algorithms are memory-based. Even though [Lam et al., 2002] seems promising, it suffers from poor search performance. This is because search using the compressed suffix array is even slower than the search using

basic suffix array by O(log *n*) factor [Sadakane and Shibuya, 2001]; *basic* suffix array in itself is slower than the suffix tree for search.

2.3.6 String B-Tree

The String B-Tree [Ferragina and Grossi, 1999] was proposed as an external memory alternative to suffix trees and suffix arrays. It is a combination of B-trees and Patricia trees for the branch nodes that is made more effective by adding extra pointers to speed up search and update operations. They were proposed to overcome the theoretical limitations of inverted files (modifiability and atomic keys), prefix B-trees (bounded-length keys), suffix arrays (modifiability and contiguous space), compacted tries and suffix trees (unbalanced tree topology) [Ferragina and Grossi, 1999]. They support search applications similar to suffix trees, however their search performance is same as B-Trees. Even though it has the same performance as regular B-trees, String B-trees handle unbounded-length strings. Figure 10 taken from [Ferragina and Grossi, 1999], shows the String B-tree for the example dataset, D = {aid, atom, attenuate, car, cod, patent, zoo, atlas, sun, by, fit, dog, ace, lid, cod, bye}, that are stored in the external memory.

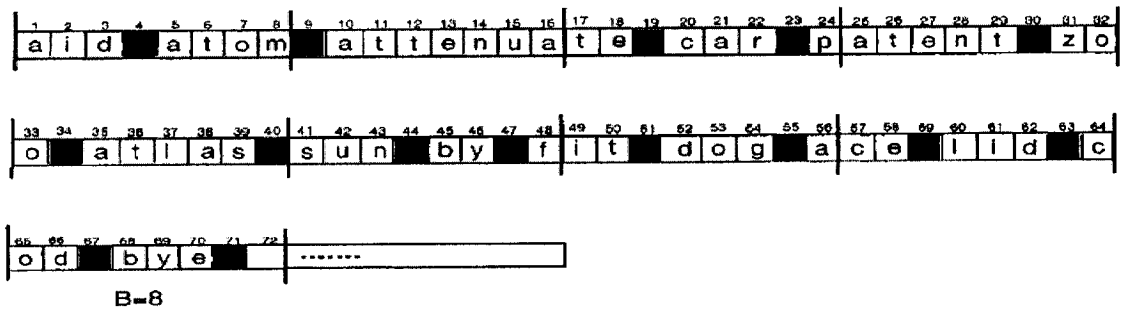


Figure 10 Sequences Stored in External Memory

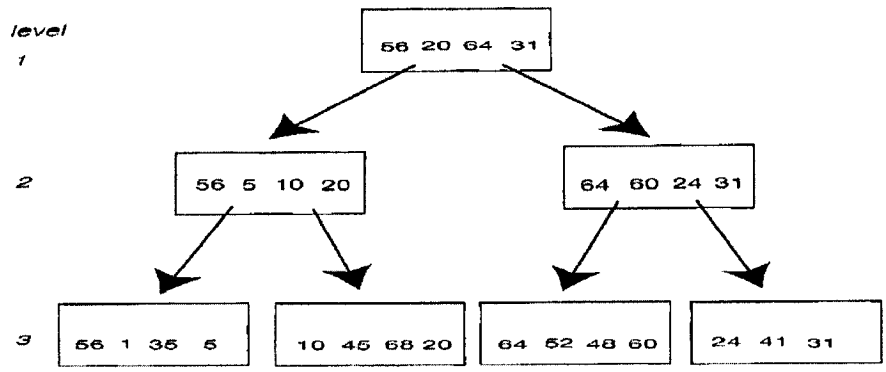


Figure 11 String B-Tree

In the figure the external memory is represented as a linear array of page size 8 bytes. The logical pointers to the individual sequences of the database *D,* represents the starting locations of each of the sequences. For example, 35 is the logical pointer to the sequence "atlas" and 69 is the logical pointer to the sequence "ye". The string B-tree index for the given sequences is shown in Figures 11, which is of height 3 and each number in the tree represents the logical pointers to the sequences. The sequences are stored in the leaves of the tree. In order to improve the search efficiency of string B-tree, each node has been implemented as a PATRICIA trie.

While string B-tree is appropriate for natural texts, it cannot be used as an indexing technique for biological sequence data. This is due to the inherent lack of structure in genetic sequences [Hunt et al., 2001].

2.4 Discussion

In the previous sections we discussed various existing indexing techniques for sequence data. These include Inverted files, q-grams, trie data structures, suffix tree, suffix arrays, and string B-Tree. As mentioned before an important characteristic of biological sequence data is that it cannot be meaningfully broken into distinct words. Inverted files,

q-grams (for low similarity sequences) and string B-Tree are not useful in general in our context and hence are not considered any further. While PATRICIA and SPINE data structures have linear time search performance, they do not scale well for large sequences due to their large index size and hence not investigated any further.

The main goal of this thesis is to develop an efficient and scalable indexing technique on a typical desktop computer for large sequence data which has a reasonable construction time, index size, and supports efficient search for various applications. As a result we restrict our attention to suffix trees and suffix arrays. Most of the related literature that we have discussed concludes that the main advantage of suffix trees over suffix arrays is their efficient support for numerous search applications. An advantage of the *basic* suffix array over the suffix tree is its efficient memory and disk utilization. However, efforts (like ESA) to make the suffix array comparable to suffix trees has resulted in an increased stored space. While in-memory performance of ESA for small sequences is better compared to suffix tree techniques [Abouelhoda et al., 2004], for long sequences, suffix tree based method, TDD [Tata et al., 2004], is the only best solution at the time of our research.

The purpose of an index is to support efficiently various search applications and have a reasonable construction time. The cost of index construction is just incurred once and will be amortized over numerous queries using the index, assuming sequences that do not require updates. Hence, having a good search performance and supporting long sequences are the two main criteria for choosing a benchmark for our experiments. Only ESA and TDD techniques fall in this category and used in our study for comparing the performance of our proposed indexing technique. We do not consider the disk-based

suffix array construction algorithm DC3 as it has been shown to be inferior to TDD on a single processor machine [Tian et al., 2005]. Also DC3 constructs only the *basic* suffix array, which has poor search performance compared to ESA and TDD.

It is worth reminding the reader that in [Skiena, 1998] it has been observed that suffix tree data structure requires better implementations. Our work addresses this concern, since we believe that the suffix tree data structure offers some opportunities to improve the search efficiency for some emerging applications. Hence, the indexing technique that we propose is also based on suffix trees. While trying to develop an efficient indexing technique, we try to strike a balance between fast disk-based index construction (as exhibited by TDD), reasonable index size (as exhibited by the WOTD technique) and efficient search (exhibited by memory-based ESA), all on a typical desktop computer.

Chapter 3

Suffix Tree Based Techniques

In the previous chapter we introduced different data structures for indexing sequence data and analyzed their performance. The theoretical superiority of suffix trees for indexing biological sequence data has been acknowledged in the related literature. However, due to problems related to their size and construction time, suffix trees had not been used widely in practice until recently. Two research works on suffix trees [Giegerich et al., 2003] and [Tata et al., 2004] have provided an effective solution to these problems. While studying these techniques, we believed that any new technique that is developed for indexing long sequences using suffix trees can benefit from the ideas introduced in them. Our proposed technique has thus been inspired in many ways by these two works. Hence, a detailed analysis of these techniques is necessary to identify their related features, strengths and limitations. This is the focus of this chapter.

3.1 Space Efficient Suffix Tree Technique

[Giegerich et al., 2003] proposed a space efficient suffix tree technique, which we refer to as Write Only Top-Down (WOTD), which constructs the suffix tree in a top-down manner and each node once written need not be accessed again. The WOTD

representation is stored as a linear array of 4 byte integers (in 32 bit machines). As already mentioned, this is the most space efficient suffix tree representation. In the worst case, this index representation requires 12 bytes per input character and 8.5 bytes per input character on an average.

For our example sequence D = AGAGAGCTT$, a high level graphical illustration of the suffix tree is shown in Figure 12. The number in the squares enumerates the order in which the tree nodes are evaluated and recorded. Each edge in this figure is labeled with the corresponding characters from D. The numbers below each leaf node s gives the starting position in D at which the suffix indicated by the edge labels on the path from the root s can be found. The actual WOTD representation that is stored on the disk is shown in the second row of Figure 13. The first row illustrates the index locations of the suffix tree array.

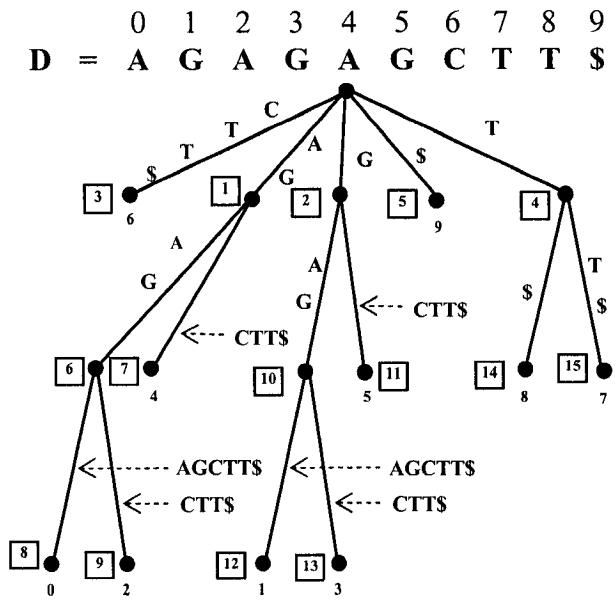


Figure 12 Suffix Tree Illustration

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	8	1	13	6	7	18	9^R	2	11	6^R	4	6^R	2	16	6^R	4	6^R	8	9^R

Figure 13 WOTD Suffix Tree Representation on Disk

WOTD Representation

We now review some definitions and the WOTD index representation taken from [Giegerich et al., 2003]. Let Σ be a finite ordered set of size k, the alphabet. Σ^* is the set of all strings over Σ, and ε is the *empty string*. For a leaf node s in a suffix tree, the leaf set of s, denoted $l(s)$, contains the position i in text D at which we can find the sequence denoted by the edge labels from the root to s. For example, for leaf node 13 (represented by the number 13 inside the square) in Figure 12, we have that $l(13)=\{3\}$. For a branching node u in a suffix tree, the leaf set of u is defined based on the leaf sets of the children of u, i.e., $l(u) = \{l(s) | \ s \text{ is a leaf node in the subtree rooted at } u\}$. For instance, for node 10 in Figure 12, the leaf set would be $l(10)=\{l(12), l(13)\}=\{1,3\}$. There is a total order $\prec$ defined on the nodes in the tree, as follows. For any pair of nodes v and w which are children of the same node u, we define $v \prec w$ if and only if $min\ l(v) < min\ l(w)$. For example, we have that node 6 $\prec$ node 10, since $min\ l(6) = 0 < min\ l(10) = 1$. For a node v, its *left pointer,* denoted $lp(v)$, is defined as $min\ l(v)$ plus the number of characters on the path from the root to the parent of v. For example, $lp(10) = 1+1=2$. A branching node u in the suffix tree occupies two adjacent suffix tree elements. The first element contains the lp value of u and two additional bits, called the *rightmost bit* and the *leaf bit*. If the rightmost bit is 1, it indicates that u is the last (w.r.t. $\prec$) node at this level. This is indicated in Figure 13 by the superscript R (for right) in the corresponding elements. For every branching node, the leaf bit is always 0. The second WOTD element which is allocated for a branching node u, stores the *firstchild* pointer. It points to the position in WOTD at which the first child node of u (w.r.t. $\prec$) is stored. For example, in Figure 13, the branching node 1 is stored in the first two elements in WOTD, i.e., WOTD[0] and

WOTD[1]. The first child of node 1 is a branching node 6 (see Figure 13). Hence, the value in the second element allocated for node 1 (i.e., WOTD[1] = 8) points to position WOTD[8], which is the first of the two WOTD elements that store node 6. The arrows above the WOTD representation in Figure 13 emphasize these pointers for illustration purposes only. A leaf node in the suffix tree occupies a single element in WOTD, in which we store the same information as in the first element allocated for a branching node: the *lp* value, the leaf bit (always set to 1), and the rightmost bit. The leaf nodes in Figure 13 are shown in grey. For the WOTD representation shown in this figure, branching node 1 is recorded in the first two WOTD elements, branching node 2 is recorded in the next two elements, leaf node 3 is recorded in WOTD[4], etc.

WOTD Construction

We now provide an overview of the WOTD construction algorithm taken from [Giegerich et al., 2003]. The WOTD construction algorithm adheres to the recursive structure of the suffix tree. The basic idea of the WOTD algorithm is that for a branching node $\bar{u}$, the subtree below $\bar{u}$ is determined by the suffixes of $D\$$ that have u as prefix. In other words, if we have the set $R(\bar{u}) := \{s \mid us \text{ is a suffix of } D\$\}$ of the remaining *suffixes* available, we can evaluate the node $\bar{u}$. First $R(\bar{u})$ is divided into groups according to the first character of each suffix. This is performed using a counting sort algorithm. For any character $c \in \Sigma$, let $group(\bar{u}, c) := \{w \in \Sigma^* \mid cw \in R(\bar{u})\}$ be the c-group of $R(\bar{u})$. If for some $c \in \Sigma$, $group(\bar{u}, c)$ contains only one sequence w, then there is a leaf edge labeled cw outgoing from $\bar{u}$. If $group(\bar{u}, c)$ contains at least two strings, then there is an edge labeled cv leading to a branching node ucv, where v is the longest common prefix (LCP)

for all the suffixes in group($\bar{u}$, c). The child ucv can then be evaluated from the set $R(ucv)$ = $\{w \mid vw \in$, group($\bar{u}$, c)$\}$ of remaining suffixes. The WOTD algorithm starts by evaluating the root from the set R(*root*) of all suffixes of D\$. All nodes of the suffix tree can be evaluated recursively from the corresponding set of remaining suffixes in a top-down manner.

For example, consider the example sequence D=AGAGAGCTT\$. The WOTD algorithm works as follows. The algorithm recognizes five groups of suffixes through a counting sort. The A-group, C-group, G-group, T-group and \$-group. Every group other than C- and \$- groups have more than one suffix. Branching nodes are created for A, G, T groups and leaf nodes are created for C and \$ groups (Figure 14).

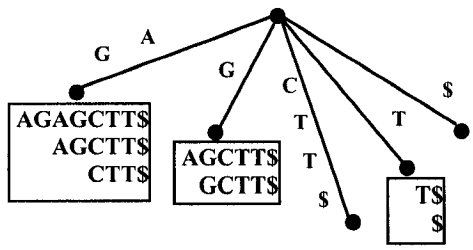


Figure 14 First Stage of Top-Down Suffix Tree Construction

The algorithm then computes the LCP of the remaining suffixes in the A-group; this common LCP is dropped from the A-group. These suffixes are then sorted using counting sort and divided into groups based on their first character. Branching nodes are created for groups with more than one suffix and leaf nodes are created for groups with just one suffix (Figure 15). Similarly all the remaining nodes are evaluated.

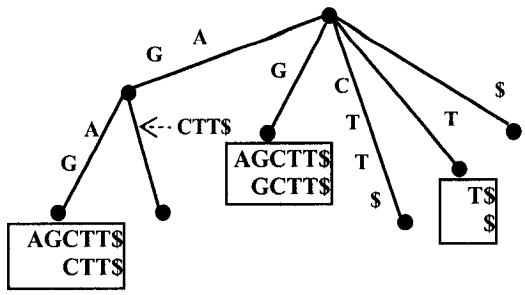


Figure 15 Second Stage of Top-Down Suffix Tree Construction

In the implementation of the algorithm, four different data structures are used, one each for the input sequence, the suffixes, temporary array (for sorting the suffixes) and the tree data structure. We refer to them as *string*, *suffixes*, *temp* and *tree*. The algorithm also uses an additional stack data structure *stack* for storing references to the unevaluated nodes. In the worst case this stack might be $n/2$ in size. In practice, however, this size is negligible compared to the other data structures and can be kept in memory.

Analysis

The WOTD construction algorithm constructs the suffix tree nodes top-down. This algorithm has worst case time complexity of $O(n^2)$. However, its good locality of reference behavior results in an expected runtime $O(n \log_\alpha n)$, where α is the alphabet size and for moderate size sequences, it is practically linear [Giegerich and Kurtz, 1995]. In [Giegerich et al., 2003] the construction time and space requirement of WOTD are compared with alternative suffix tree techniques - linked list suffix tree representation (*mccl*) [Kurtz, 1999] and hash table suffix tree representation (*mcch*) [Kurtz, 1999], both of which implement the McCreight's suffix tree construction algorithm. All three algorithms have similar average construction times for different types of sequence data;

however, WOTD and *mccl* have relatively stable run times due to the fact that they are independent of the alphabet size.

WOTD requires the smallest working space. This is expected due to its space efficient index representation. The WOTD representation requires $2q + n$ integers, where q is the number of non-root branching nodes, i.e., $q = n\text{-}1$ in the worst case. The experiments in [Giegerich et al., 2003] show that the WOTD representation requires 12 bytes per input character in the worst case and 8.5 bytes per input character on average for a collection of sequences of different types. Compared to the *mccl* and *mcch* representations, their index size is less by 0.84 and 5.54 bytes per character, respectively.

A comprehensive study of the memory-based search performance of WOTD as an index for exact match searching is also reported in [Giegerich et al., 2003]. The search experiments show its superiority compared to alternative suffix tree representations. In addition to *mccl* and *mcch*, the authors also compare the performance of WOTD representation with the augmented suffix array (*asa*) [Manber et al., 1991] and their implementation of Boyer-Moore-Horspool algorithm (*bmh*) [Horspool, 1980]. The search using WOTD is faster by an order of magnitude compared to using the *asa* representation and has an advantage in search using the *bmh* algorithm. While the search performance using WOTD, *mccl*, and *mcch* are comparable, the last two representations are not suitable for indexing large data sequences in external memory. Even though the WOTD construction algorithm proposed and implemented in [Giegerich et al., 2003] is memory-based, [Tata et al., 2004] has shown, using a variant of the WOTD representation, that the WOTD technique can be adapted for disk-based suffix tree construction algorithms over

long sequences. Further, extension of the WOTD technique to the disk is one of our contributions in this thesis.

This WOTD representation however has the following limitation. The *lp* value, which is stored in the first WOTD element for a branching node (and in the single element for a leaf node), has a value in the range [0...*n*], where $n = |D|$. Since 2 bits are reserved for the *rightmost bit* and the *leaf bit*, there are only 30 bits left for storing the *lp* value, when we use 4 byte integers for the suffix tree elements. This limits the size $|D|$ of the sequence data that can be indexed using this representation to 2^{30}-1, or about 1 billion symbols. At first, this limitation may seem to have no practical concern, but not when dealing with biological sequence data. For example, the human genome includes more than 3 billion symbols.

3.2 Top-Down Disk-Based (TDD)

Most suffix tree construction algorithms do not scale well due to the poor locality of reference exhibited by their data structures. The TDD technique was proposed in [Tata et al., 2004] as a practical way to construct suffix trees for very long sequences. The main contribution of this technique is that it enabled the construction of suffix trees for sequences as large as the human genome (approximately 3 billion symbols).

Proposed in [Tata et al., 2004] is a construction algorithm called PWOTD (Partition and Write Only Top-Down). This is an extension of the WOTD construction algorithm with a partitioning phase (proposed in [Hunt et al., 2001]) and the proposed buffer management

strategy. The TDD index representation which is a variant of the WOTD representation is explained next followed by a description of the PWOTD construction algorithm.

TDD Representation

The underlying concept of the TDD index representation is the same as that of WOTD. As discussed in Section 3.1, the WOTD has a disadvantage that it cannot index sequences larger than 1 billion symbols. This limits the use of the WOTD technique when dealing with enormous amounts of data which is typical in the biological domain.

In order to overcome this limitation on the size of the sequence, the TDD technique adopts an alternative approach for the index representation. The TDD technique introduces two additional bitmap arrays, one bitmap array for storing the leaf bits and the other for storing the rightmost bits. Due to the introduction of two separate bitmap arrays, for each node in the suffix tree, all 32 bits are available for recording its *lp* value. Similar to the WOTD representation, a branch in TDD is represented using two suffix tree elements and a leaf is represented using one suffix tree element. We refer to this representation as the TDD representation. Figure 16 shows the structure of both the branch and leaf nodes in the TDD representation of the suffix tree.

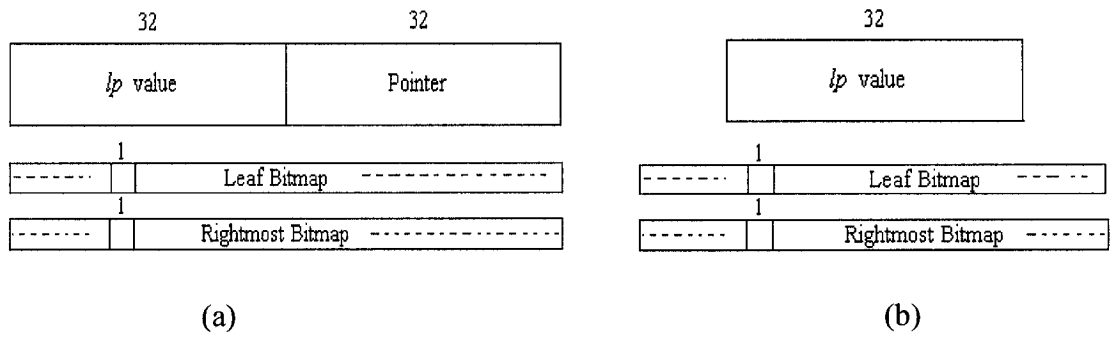


Figure 16 TDD Index Representation (a) Branch Node (b) Leaf Node

While this solution works fine for sequences larger than 1 billion symbols, it may result in slower construction times and search performance compared to WOTD due to the presence of additional data structures. In Section 4.4 we experimentally show that this is in fact the case.

PWOTD Construction Algorithm

The PWOTD algorithm proceeds in the same way as the WOTD algorithm introduced in the previous section. The difference is the additional partitioning phase in the PWOTD algorithm. The PWOTD algorithm [Tata et al., 2004] is presented in Figure 17.

Algorithm PWOTD(*Sequence, prefixlen*)
Phase1:
Scan the *Sequence* and partition *Suffixes* based
on the first *prefixlen* symbols of each suffix
Phase2: For each partition, do the following
1. START BuildSuffixTree
2. Populate *Suffixes* from current partition
3. Sort *Suffixes* on first symbol using *Temp*
4. Output branching and leaf nodes to the *Tree*
5. Push the nodes pointing to an unevaluated range
onto the *Stack*
While *Stack* is not empty
6. Pop a node
7. Find the Longest Common Prefix (LCP) of
all the suffixes in this range by checking
the *Sequences*
8. Sort the range in *Suffixes* on the first
symbol using *Temp*
9. Write out branching nodes or leaf nodes to *Tree*
10. Push the nodes pointing to an unevaluated range
onto the *Stack*
END While
END of PWOTD

Figure 17 PWOTD Construction Algorithm

In the first phase of the algorithm, it partitions the suffixes in order to handle large sequences. The idea of partitioning is that, once the suffixes are partitioned based on the first *prefixlen* characters, each partition can be loaded in the main memory and their corresponding subtrees can be built independent of the others. For example, consider our

example sequence *D*=AGAGAGCTT$. Using the PWOTD algorithm, with a prefix length (*prefixlen*) of 1, we obtain four partitions (we ignore the partition for $). The suffix partition for A would be {0, 2, 4}, representing the suffixes {AGAGAGCTT$, AGAGCTT$, AGCTT$). Similarly the suffix partition for C would be {6}, representing the suffix {CTT$}, and so on. During the second phase the WOTD algorithm introduced in Section 3.1 is used to construct the suffix trees corresponding to each of these partitions, independently.

Analysis

The TDD technique was introduced as an efficient and scalable indexing technique for large sequences and is an extension of the WOTD technique. The partitioning phase in TDD can be performed in O(n) time. As a result, the worst time complexity of the construction algorithm remains unaffected and is O(n^2). It has been shown that due to its better locality of reference the TDD technique outperforms the memory based linear time Ukkonen algorithm on modern cached architectures [Tata et al., 2004]. TDD has also been shown to be superior to both TOP-Q [Bedathur and Harista, 2004] and DC3 [Dementiev et al., 2005].

In order to handle sequences larger than 1 billion symbols, TDD introduces two additional bitmap arrays to store the leaf and right most bits. This results in an additional storage overhead. For example, the total size of the bitmap arrays for human chromosome 5 (169 million symbols) is 550 MB and the index size is around 1600 MB.

While TDD has been shown to be superior to alternative techniques in terms of construction time and ability to handle large sequences, the search performance using

their index is not studied. In Section 4.4 and Section 5.2, we experimentally evaluate the search performance of TDD.

3.3 Summary

In this chapter we introduced two suffix tree based indexing techniques, WOTD and TDD, both of which inspired our work. The WOTD technique is memory-based and can support sequences as large as 1 billion symbols, provided there is enough available RAM. The advantage of the WOTD technique is its space efficiency and locality of reference for the tree data structure, since once a tree node is evaluated it need not be visited again during the construction of other parts of the tree.

The TDD technique extends WOTD to a disk-based technique and overcomes the 1 billion symbols limitation. This is done by implementing efficient buffering strategies that make use of the focused locality property of the WOTD technique and introducing two additional bitmap arrays. While the addition of bitmap arrays results in a scalable solution that can handle sequences as large as 4GB (on 32 bit computers), it introduces an overhead during the construction and search process, as separate buffering of these arrays are required and this requires additional memory. In fact we noticed that in the TDD implementation which was made available to us, the entire bitmap array is kept in the memory throughout the suffix tree construction.

While TDD has been shown to be superior to alternative suffix tree and suffix array based construction algorithms, its search performance has not been studied. In particular we are curious about the effect of the additional bitmap arrays on the TDD construction

and search algorithms compared to WOTD. This study in our experiments resulted in a better understanding of the influence of the index representation on its performance, for both construction and search algorithms, which resulted in our improved suffix tree based techniques, presented in the following chapters.

Chapter 4

Our First Experience

Suffix tree, being the most versatile solution for indexing biological sequence data, is the chosen indexing structure for our work. In the previous chapter, we introduced two efficient suffix tree based implementations, TDD and WOTD, which inspired our work. The main objective of this thesis is to develop an efficient and scalable indexing technique for sequence data. Even though TDD seems to be an excellent starting point for our work, there is also a need to investigate the efficiency of the TDD technique compared to WOTD technique. A comparison of these index representations have not been done before in related literature and the influence of the chosen index representation on the construction and search is not well understood. Proper understanding of these techniques will be crucial in the design of a new and efficient indexing technique.

As a first step in this direction, we extend the WOTD technique to the disk by implementing the corresponding disk-based algorithms. We refer to our disk-based extension of the WOTD technique as Suffix Tree Top-Down 32 bit (STTD32) technique. An important aspect that should be further investigated is the impact of the order in which the suffix tree nodes are recorded on the performance. Recall that both WOTD and TDD evaluate and record the nodes in top-down order. Their corresponding search algorithms perform a top-down traversal of the suffix tree, in order to achieve better

performance. However, for some search applications that use the index, performing a depth-first traversal of the tree for retrieving the sequence data might be beneficial as a depth-first traversal uses less memory compared to the top-down traversal. In such cases, combining depth first traversal with top-down order of the suffix tree nodes may result in poor performance. Hence, it is worth investigating if a depth first ordering of the nodes would be beneficial and this is the reason why we developed our Suffix Tree Depth-First 32 bit (STDF32) technique [Halachev et al., 2005].

In the rest of this chapter, we present the above mentioned extensions of WOTD that we develop and compare the performance of TDD, STTD32, and STDF32 indexing techniques based on their construction time, storage requirements, and search time, using various types of real life sequence data.

We acknowledge the fact that both the STTD32 and STDF32 cannot handle sequences larger than 1 billion symbols (Section 3.1). However, this study will shed more light on the various sources that influence the performance of the alternative techniques which we will use to identify the essential characteristics of our desired indexing technique(s) for handling large sequences.

4.1 Alternative Top-Down Technique (STTD32)

One of our contributions is extending the WOTD technique to the disk-based computational model. This is an alternative top-down technique that we have

implemented for the purpose of our comparison study with TDD. We shall refer to this as the Suffix Tree Top-Down 32 bit (STTD32) technique. The 32 bit in the name signifies that each element in the suffix tree is of size 32 bits (4 bytes).

We implemented the STTD32 technique by first implementing the WOTD construction algorithm, described in Section 3.1. To this algorithm we added a partitioning phase that was proposed by [Hunt et al., 2001]. In order to have a fair comparison between the alternative index representations, we implemented the buffering strategies proposed and implemented in TDD [Tata et al., 2004]. The resulting construction algorithm is similar to the one implemented in TDD (Figure 17). The difference between STTD32 and TDD is in the final representation of the suffix tree and the buffering strategy they use. Our index representation is the same as proposed by [Giegerich et al., 2003], while TDD introduces two additional bitmap arrays (refer Section 3.2). We further improved the buffering strategy proposed in [Tata et al., 2004] by considering and implementing a dynamic buffering strategy.

As a part of our research work, we also implemented the corresponding exact match search algorithms and proposed their associated buffering strategies. The study of these search algorithms is interesting topic, in itself but not discussed further in this thesis. A detailed description of the search algorithms and their performance evaluation can be found in [Halachev et al., 2005] and [Halachev et al., 2007].

Analysis

The STTD32 representation is the same as the WOTD representation and hence has the same space requirements, i.e., 12 bytes/character in the worst case and 8.5 bytes/character

on average [Giegerich et al., 2003]. Even though we have added an additional partitioning phase to the WOTD construction algorithm, it does not affect the $O(n^2)$ time complexity. This is because the partitioning can be performed through one read of the input sequence, done in $O(n)$ time.

4.2 Depth-First Technique (STDF32)

The effectiveness of an index is estimated by its support for various searching techniques. The motivation for an alternative depth-first technique can be explained better by first looking at the search applications that use the suffix tree index. A depth-first traversal of the suffix tree to retrieve the answer nodes is preferred, in some search applications as it uses less memory compared to the top-down traversal. This additional memory can be used for storing the large number of intermediate nodes during the search process. Depth-first traversal becomes even more crucial when considering sequences over larger alphabets like proteins and natural texts where the suffix trees have larger fan outs.

While our ultimate research goal is not to provide two different index structures for the different search applications, it would be beneficial to study which ordering of the nodes results in improved performance by offering better locality of reference during various search operations. The information obtained in this study can be considered and used in our solution for handling very large sequences.

STDF32 Index Representation

For our example sequence D = AGAGAGCTT\$, the numbers in the square in the suffix

tree graphical representation in Figure 18 show the order in which the suffix tree nodes are evaluated and recorded by the construction algorithm, for STDF32.

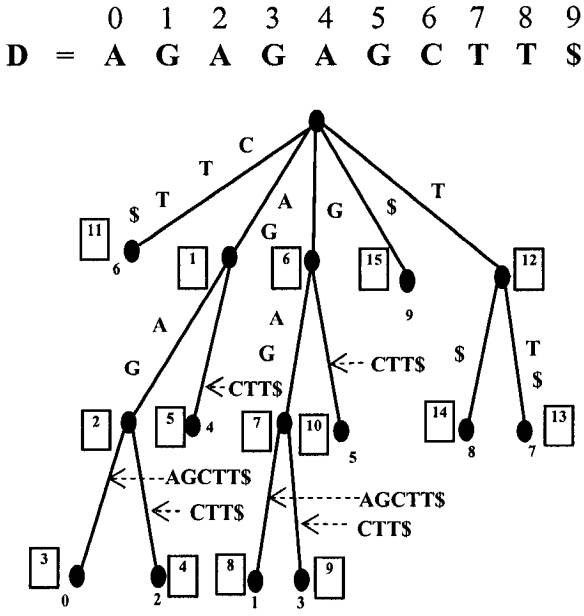


Figure 18 Depth-First Tree Traversal

In the above figure, each edge is labeled with the corresponding characters from *D*. The numbers below each leaf node *s* gives the starting position in *D* at which the suffix indicated by the edge labels on the path from the root *s* can be found. The actual index STDF32 representation on disk is shown in Figure 19. The Figure uses the same illustration convention as used for STTD32 shown in Figure 13.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	7	2	6	4	6^R	6^R	1	14	2	13	4	6^R	6^R	6	7	19	8	9^R	9^R

Figure 19 STDF32 Index Representation on Disk

The major difference between the STTD32 and STDF32 representations is in the order in which the nodes are evaluated and stored in the suffix tree. Similar to the STTD32 representation, the depth-first version requires two suffix tree elements to represent a branching node and one element to represent the leaf. In Figure 19, the first two elements

record the branching node 1, the next two elements record the branching node 2, the fourth element records the leaf node 3, etc. Since STDF32 is a depth-first technique, it is always the case that the *firstchild* of a branching node, is stored immediately after the second element allocated to the current branching node. For example, the *firstchild* of node 1 (the first branching node stored in STDF32) is node 2, the *firstchild* of node 2 is node 3, and they are all stored in adjacent array locations. Since we know the location of the *firstchild* node implicitly, we use the pointer in the second element for the branching node in the STDF32 representation to store a pointer to the *rightsibling*. For example, in Figure 19, branching node 1 has node 6 as its *rightsibling*. Hence in the pointer element for branching node 1, we store the position in STTDF32 where node 6 is recorded (i.e. position 7). The other information stored in the two elements in STDF32 is the same as that in STTD32. For leaf nodes, we store the same information as in STTD32. It should be noted that the changes introduced in STDF32 do not result in any change in the number of array elements.

STDF32 Construction Algorithm

The construction algorithm for the STDF32 technique, referred to as Partition and Write Only Depth-First (PWODF) is shown in Figure 20. It is a variant of the PWOTD construction algorithm explained in Section 3.2. The difference between the two algorithms is in the order in which the nodes are evaluated and stored. Similar to the PWOTD construction algorithm introduced in Section 3.2, PWODF also has a partitioning phase.

Algorithm PWODF (*sequence D, prefixlen*)
Phase1:
Scan D and partition *Suffixes* based on the first *prefixlen* symbols of each suffix
Phase2: For each partition do the following:
1. Populate *Suffixes* from current partition
2. Sort *Suffixes* on first symbol using *Temp*
3. Push unevaluated *Suffixes* range.
4. While *Stack* is not empty
5. Pop the unevaluated *Suffixes* range.
6. Find the Longest Common Prefix (LCP) of all the *Suffixes* in this range by checking *D*
7. Sort *Suffixes* on the first symbol using *Temp*
8. If a leaf node is encountered output it to the *Tree*. Repeat Step 8 for the remaining *Suffixes* in the range.
9. Else if a branching node is encountered, process the node as follows:
10. Push the remaining unevaluated *Suffixes* range onto the *Stack*.
11. Output current branching node to the *Tree*.
12. Push the current branching node pointing to an unevaluated range onto the *Stack*.
13. END While
END PWODF

Figure 20 PWODF Construction Algorithm

For example, consider our example sequence *D*=AGAGAGCTT$. A memory-based STDF32 construction algorithm works as follows. The algorithm recognizes all four groups of suffixes through a counting sort. The A-group, C-group, G-group, T-group, and $-group. The first group which has the suffix starting at the least significant sequence index is chosen (w.r.t. $\prec$). The range of *suffixes* that are unevaluated are pushed onto the stack. If this group contains only one suffix, a leaf node is created; otherwise a branching node is created. In our example the group-A contains the suffix which starts at the least significant location in the sequence (at index 0). A branching node is created for this suffix. This is shown in Figure 21.

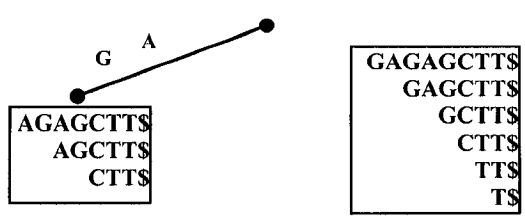


Figure 21 First Stage of Depth-First Suffix Tree Construction

The algorithm then computes the LCP of the remaining suffixes in A-group, and this common LCP is dropped from the group. These suffixes are then sorted using counting sort and divided into groups based on their first character. The algorithm then constructs the suffix tree by repeating the above process. The second stage of the algorithm is shown in Figure 22.

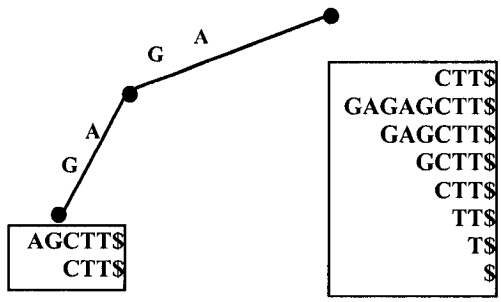


Figure 22 Second Stage of Depth-First Suffix Tree Construction

For the disk-based construction, the algorithm first partitions the suffixes based on the first *prefixlen* characters. Then the construction of the suffix tree for each partitions proceeds as explained above. It can be seen from the example that during the construction of the tree in depth-first order, the space required for the *stack* is more than STTD32. This is because during the top-down construction of the tree, the siblings of a particular node are created before evaluating the child nodes. These unevaluated sibling nodes can be used to store the suffixes boundary that needs to be processed during their evaluation. For example in Figure 18, during top-down construction, before evaluating the children of node 1, the sibling nodes 6, 11, 12, and15 are created. In the case of branch nodes 6 and 12, the information on the suffixes that needs to be processed to build their corresponding subtrees i.e. boundary of the suffixes array, is stored in the second cell of the unevaluated branching nodes. However, in the case of a depth-first traversal, the child nodes are created and evaluated first and information about the suffixes boundary

are stored in the stack in addition to the unevaluated nodes. This additional information is in the form of pointers to the left and right boundaries of the suffix that need to be evaluated. The stack for the depth-first construction has been implemented as a structure array. Maintaining this stack results in slightly slower construction time especially when the alphabet size is small and the depth of the tree is large, as in the case with DNA data (discussed in Section 4.4.1). An advantage of the depth-first traversal is that, since, there are no unevaluated tree nodes created (unlike STTD32), the tree buffer is used more efficiently. This is shown experimentally and discussed in Section 4.4.1.

Analysis

STDF32 differs from STTD32 only in the order in which the nodes are evaluated and created. The new representation does not have any additional storage cost and hence has the same space requirements as STTD32. The STDF32 construction algorithm has the same $O(n^2)$ complexity as STTD32.

4.3 Buffering Strategy

We now explain in detail the buffering strategy proposed in [Tata et al., 2004], which we adopted in the construction algorithms of all our techniques. Suffix trees are an order of magnitude larger in size than the input sequence. Hence, during the construction of the suffix tree, especially for large sequences, we need more main memory. For example, the suffix tree for the human chromosome 2 is around 2 GB, for which and other such large sequences, it is essential to perform buffering of data.

As already explained in Section 3.1, during the construction of the suffix tree, four different data structures are required – *string*, *suffixes*, *temp* and *tree*. Figure 23 gives an overview of the buffering strategy and the relative size of these different data structures in memory and on disk, taken from [Tata et al., 2004].

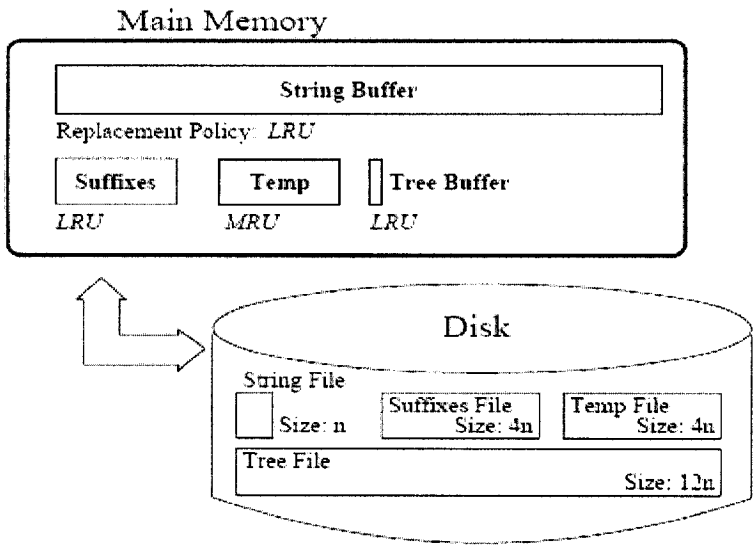


Figure 23 Buffering Strategy

The *tree* has the highest locality of reference. It is an order of magnitude larger than the sequence itself. However, once the nodes are evaluated they are never visited again and hence the accesses are mostly sequential in nature. This access pattern displays good temporal and spatial reference. As a result, the *tree* buffer is implemented using a simple LRU (Least Recently Used page replacement policy).

Accesses to the *suffixes* array are mostly sequential. During the counting sort, the *suffixes* are traversed sequentially and copied to the *temp*. And after sort, there are random writes from the *temp* back to the *suffix*. However, these random writes occur at the character group boundaries and occur sequentially to their right. This minimal locality of reference exhibited by the *suffixes* array allows us to use LRU for the *suffixes* buffer. *Temp* array is

referenced using two linear accesses during the construction process. Hence a MRU (Most Recently Used replacement policy) is preferred for *suffixes* array.

The *sequence* array is the smallest of the four arrays but is referenced at random locations when performing the counting sort and finding the longest common prefix in each sorted group. As a result of this, it has the least locality of reference compared to the other data structures. It is therefore advisable to allocate as much memory as possible to the sequence array. If there is not enough memory, LRU replacement policy can be used. When the size of the sequence is larger than the available main memory, the sequence is compressed by packing each character symbol into 4 bits. This enables us to maintain a larger part of the sequence in memory.

The buffer size of each of these data structures is allocated based on the input sequence size and the available main memory. A more detailed study on the buffering scheme is reported in [Tata et al., 2004]. We extended this proposed buffering policy, by implementing a dynamic buffering strategy. In their buffering strategy, the sizes of the buffers are determined at the beginning of the algorithm. If a partitioning phase is performed, then the buffering of the *suffix* and *temp* data structures are performed based on the size of the largest partition. This adds an additional buffering overhead when most of the partitions fit in memory. To avoid this buffering overhead, in our techniques (STTD32 and STDF32), the buffering of all data structures is determined at run time depending on the size of the partition that is currently being evaluated. Hence, when the whole partition fits in the main memory, we do not incur any additional buffering overhead.

4.4 Performance Evaluation

In this section we present a thorough analysis of all the three suffix tree based indexing techniques – STTD32, STDF32, and TDD based on their index size, construction time, and memory buffer usage. We also evaluate the disk access patterns of STTD32 and STDF32 in order to determine which ordering of the nodes would be useful. A comparison based on their search performance is also provided. All considered techniques are disk-based. TDD can handle sequences as larges as 4GB (on 32 bit machines), while STTD32 and STDF32 can handle sequences as large as 1 billion symbols (1GB).

Experimental Settings All our experiments were conducted on a standard desktop machine running Linux Kernel 2.6.19 with Intel Pentium 4 @ 3 GHz clock speed, 2 x 160 GB HDD (7200 RPM), 2 GB RAM, and 2MB L2 cache. STTD32 and STDF32 programs were written in C and compiled using *gcc* with all optimizations (-O3) enabled. The TDD source code was obtained from Tata's website which was written in C++ and compiled using g++ with all optimizations enabled. For all the three techniques we consider buffer pages of size 8192 bytes (8KB).

Experimental Datasets In these experiments we consider the following sequences (1) DNA sequences – Human Chromosome 3, 5, 10, 15, 20, and Y; (2) Protein Data (Swiss-Prot [SProt, 2005]); and (3) Natural texts from Project Gutenberg [Guten, 2005]. We preprocessed the input DNA sequences by removing symbol N, denoting the unknown symbol and thus the DNA alphabet size is 4. Figure 24 shows the sizes of the chromosomes we used in our experiments. For Swiss-Prot database, we removed the

header lines, new line symbols, and blanks. The resulting sequence, called *sprot*, is about 75 million characters over an alphabet of 23 symbols. The natural text sample was cleaned by removing non-printable symbols. The resulting sequence, called *guten80*, contains the first 80 million characters taken from [Guten, 2005] over an alphabet of 95 symbols.

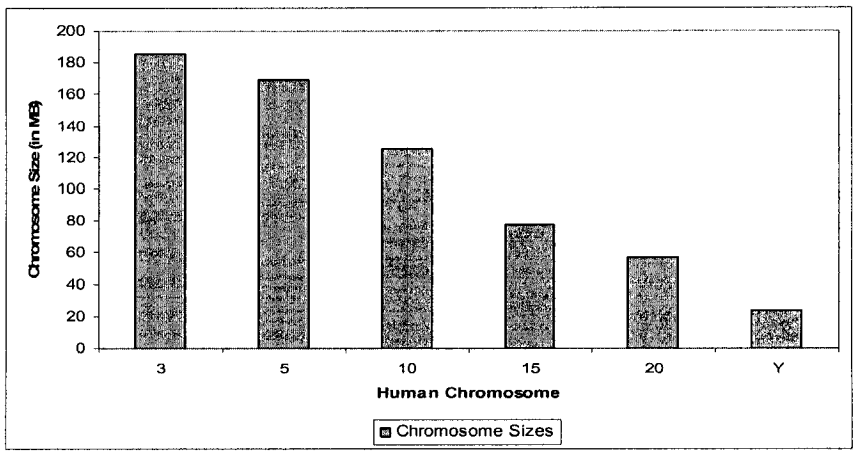
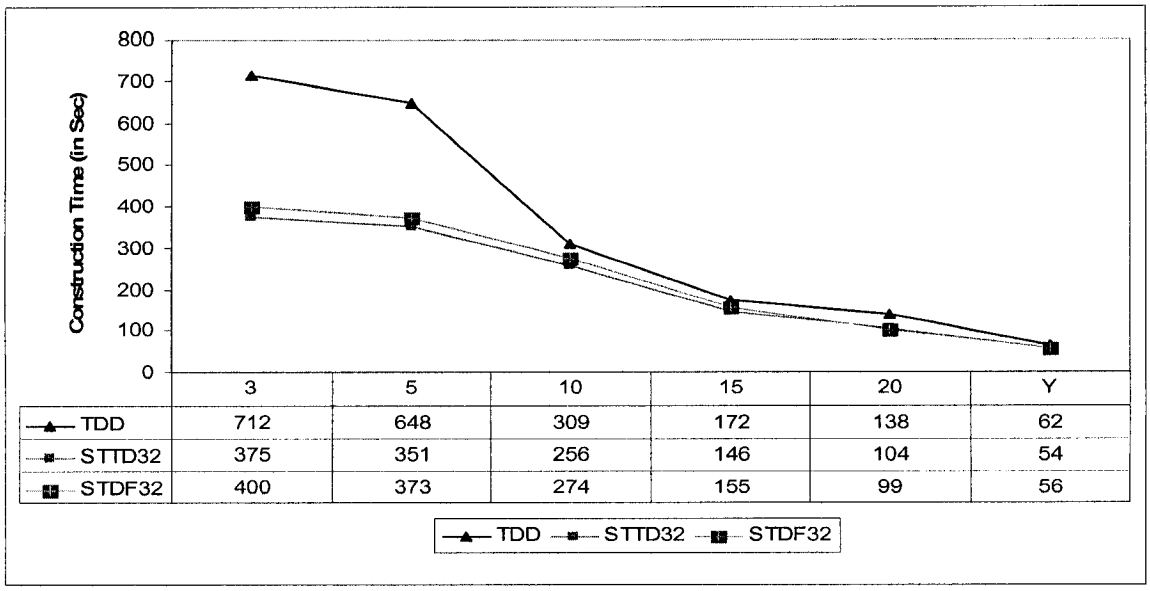


Figure 24 Chromosome Sizes

4.4.1 Index Construction Time and Size

In this set of experiments, we compare the performance of the indexing techniques based on their index construction time, index size. We also study the influence of the alphabet size on their index performance. Figure 25 shows the construction times of the considered techniques using the DNA sequence data.



	3	5	10	15	20	Y
TDD	712	648	309	172	138	62
STTD32	375	351	256	146	104	54
STDF32	400	373	274	155	99	56

Figure 25 Chromosome Index Construction Times for TDD, STTD32 and STDF32

It can be seen from figure that creating the nodes in the depth-first order has very little effect on the construction time. This is because even though the nodes are recorded in a different order, the construction algorithms of both STTD32 and STDF32 need to evaluate and create the same number of nodes and this is done using almost the same number of instruction cycles using almost very similar amounts of memory. STDF32 is slightly slower than STTD32 due to the fact that it requires maintaining and storing additional information in the *stack* during the index construction, as discussed in Section 4.2. Unlike in the case of STTD32, where we create the unevaluated siblings of a node before evaluating its children, no extra unevaluated tree nodes are created for STDF32, and hence the tree buffer is utilized more efficiently. This can be seen by the relatively low tree buffer I/O for STDF32 in Figure 26.

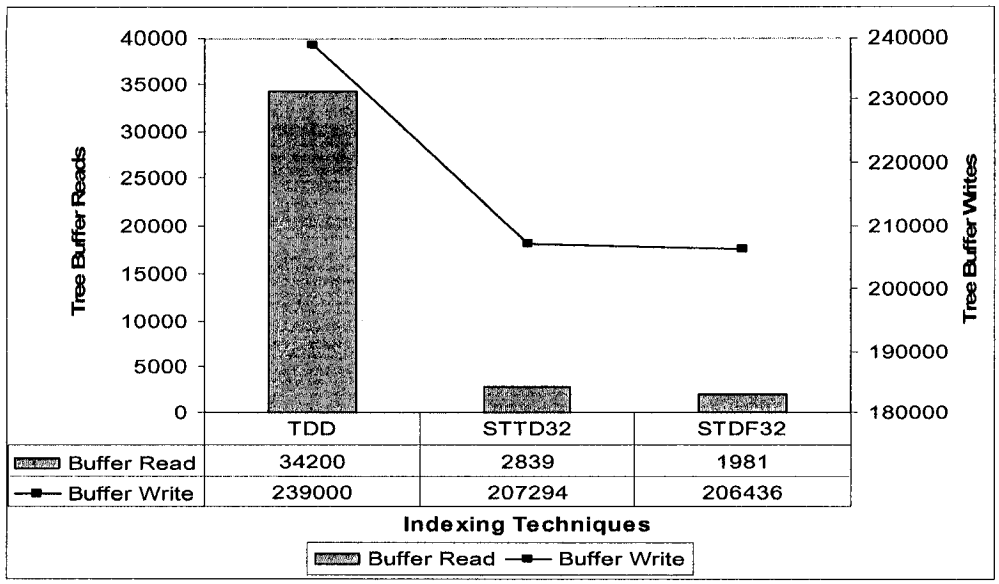


Figure 26 Tree Buffer I/O for TDD, STTD32 and STDF32 (Chromosome 5)

It can be seen from Figure 25 that both our techniques STTD32 and STDF32 outperform TDD for all considered DNA sequences. This difference is less obvious for smaller sequences such as Chromosome Y while it is more obvious for large sequences like Chromosomes 3 and 5. This improvement in construction time can be attributed to better memory utilization by our techniques compared to TDD, where additional data structures are used to store the leaf and rightmost bits. As a result of using bitmap arrays, there will be less memory available for other data structures in the memory, which results in slower construction time. The effect of the additional data structures on the index construction is shown more clearly in Figures 26 and 27. Figure 26 shows the *tree* buffer I/O for Chromosome 5. In both STTD32 and STDF32, 64 memory pages are allocated to the tree, while in TDD only 8 pages are allocated. Since STTD32 and STDF32 do not require any additional data structures, more space is available and hence allocated to the *tree*, resulting in less *tree* buffer I/O. For all the techniques and for all considered human chromosomes, the *suffixes* partition and *string* can be kept in memory. Hence no buffering is required. The absence of the leaf and rightmost bitmap arrays in our indexes

enables us to utilize the available memory for the *temp* data structure and hence avoid buffering it. For TDD, however, the *temp* data structure for chromosome 5 is buffered as extra memory needs to be allocated for the bitmap arrays. For *temp*, TDD requires an additional $1.6x10^6$ disk I/O operations. The disadvantage of using the bitmap arrays becomes even more obvious considering the percentage of space required for each of these data structures. This is shown in Figure 27.

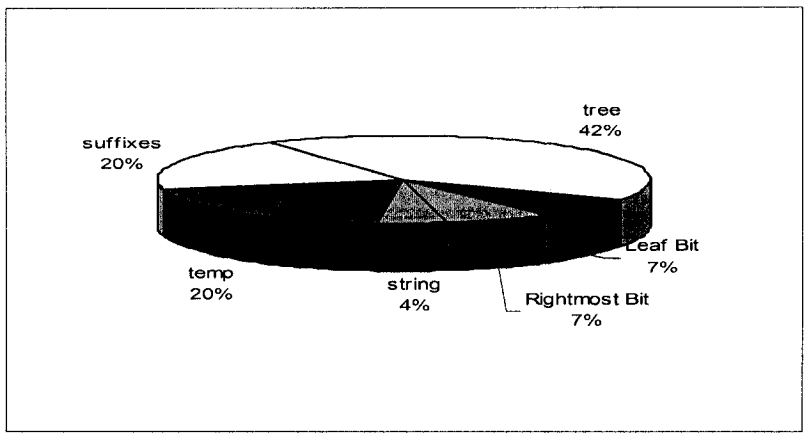


Figure 27 Space Requirements of Different Data Structures for Chromosome 5

It can be seen in the above figure, that the leaf and rightmost bit arrays, which are kept in memory throughout the TDD construction process, require a considerable amount of memory (about 550MB for Chromosome 5, which is significant compared to the 2GB size of our available memory). As a result, the memory available for the other data structures is greatly reduced in TDD compared to our techniques, which leads to buffering of *temp* and *tree* in the case of TDD technique.

Effect of the Alphabet Size

Figure 28 shows the construction times for proteins ($|\Sigma| = 23$) and natural text ($|\Sigma| = 95$) for all the three disk-based suffix tree indexing techniques considered.

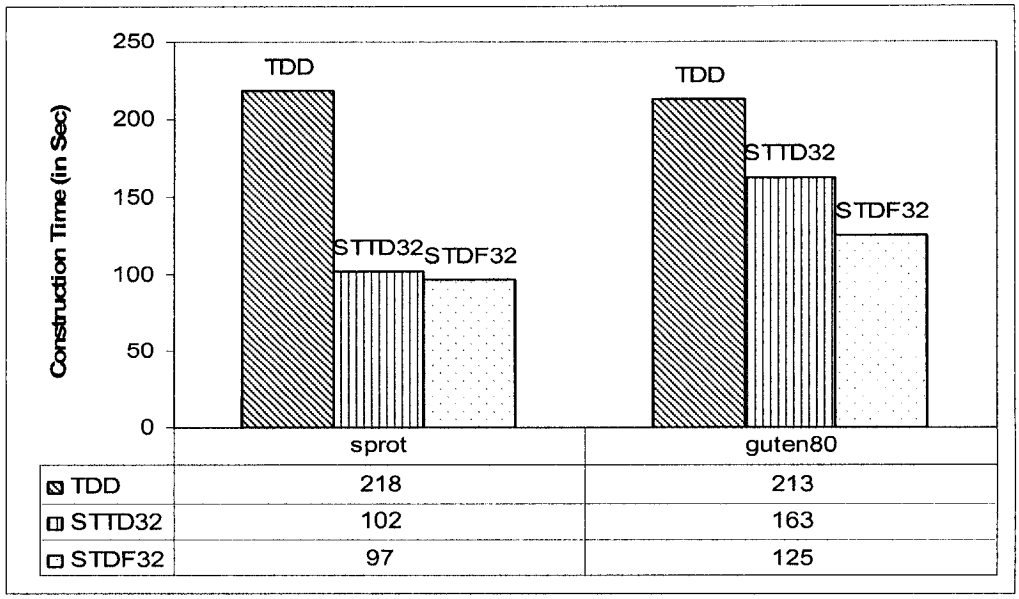


Figure 28 Construction Times for Protein and Natural Text Sequences

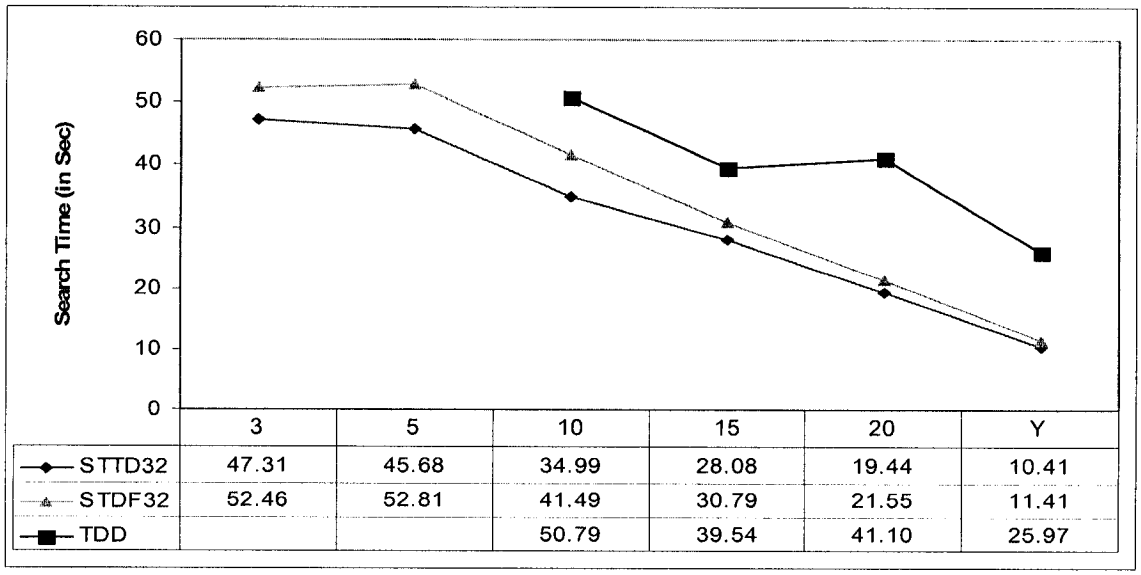
TDD has the slowest construction times for these types of data and STDF32 is the fastest of the three. This is because when the alphabet size is large, the depth of the tree is small compared to the breadth. As a result, there is less additional information to be maintained in the depth-first construction algorithm. This combined with the locality of reference exhibited by the depth-first tree (see Figure 28) results in faster construction times.

Index Size

STTD32 and TDD require $9.5n$ and $12.7n$ bytes per character on average for DNA data. The space required for the suffix tree for *sprot* ($|\Sigma| = 23$) for STTD32 and TDD reduces to $8.5n$ and $11.8n$, respectively. This is because the larger branching factor leads to less number of branching nodes and hence smaller index size. Similarly for guten80 ($|\Sigma| = 95$), the storage requirements for STTD32 and TDD are $8.1n$ and $11.4n$, respectively. STDF32 has the same space requirements as STTD32.

4.4.2 Exact Match Search Performance

In this experiment, we evaluate the indexing techniques based on their exact match search (EMS) performance. To compare the search times for the considered DNA sequences, we sampled Chromosome 2 and obtained three sets of 1200 queries, each of length 7, 11 and 15. The resulting query files are referred to as 7_1200q, 11_1200q and 15_1200q. The lengths (7, 11, and 15) are chosen based on the word lengths considered in BLAST. The DNA search times, using the three indexing techniques for the query set 11_1200q is shown in Figure 29. The search times for the query sets 7_1200q and 15_1200q are similar to those reported for 11_1200q.



	3	5	10	15	20	Y
STTD32	47.31	45.68	34.99	28.08	19.44	10.41
STDF32	52.46	52.81	41.49	30.79	21.55	11.41
TDD			50.79	39.54	41.10	25.97

Figure 29 Exact Match Search Time for STTD32, STDF32 & TDD

As it can be seen from the above figure, STTD32 provides the most efficient EMS performance. The depth-first ordering of the nodes does not provide an advantage to STDF32 technique and it is slower compared to STTD32. As we suspected, TDD provides the slowest search time due to the presence of additional bitmap and leaf bit arrays that need to be accessed in addition to the tree data structure. The EMS algorithm

provided with TDD is memory-based and hence it was not able to search in large sequences (Chr3 and Chr5).

The search performance of the indexing techniques on *sprot* and *guten80* are shown in Figure 30. The query set used for *sprot* is obtained by sampling the entire Swiss-Prot database [SProt, 2005]. It contains 500 query words of size 2 characters and 500 words of size 3 characters. Again, the query word sizes are chosen accordingly to word sizes used by BLAST. The query set used for *guten80* contains the 100 most frequent words found in the Gutenberg collection [Guten, 2005], and each word is of size greater than 3 characters.

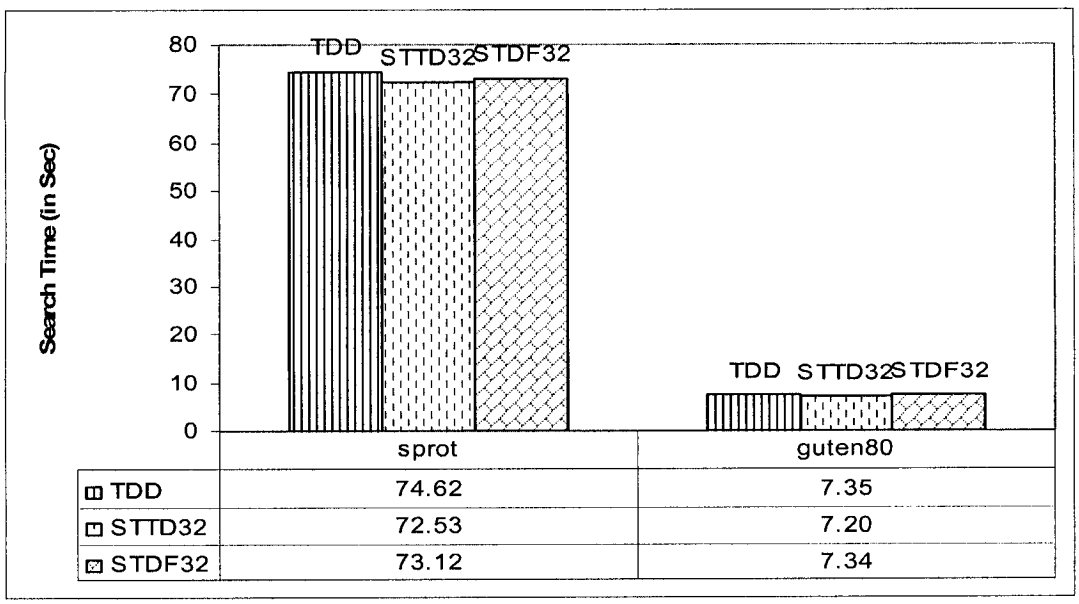


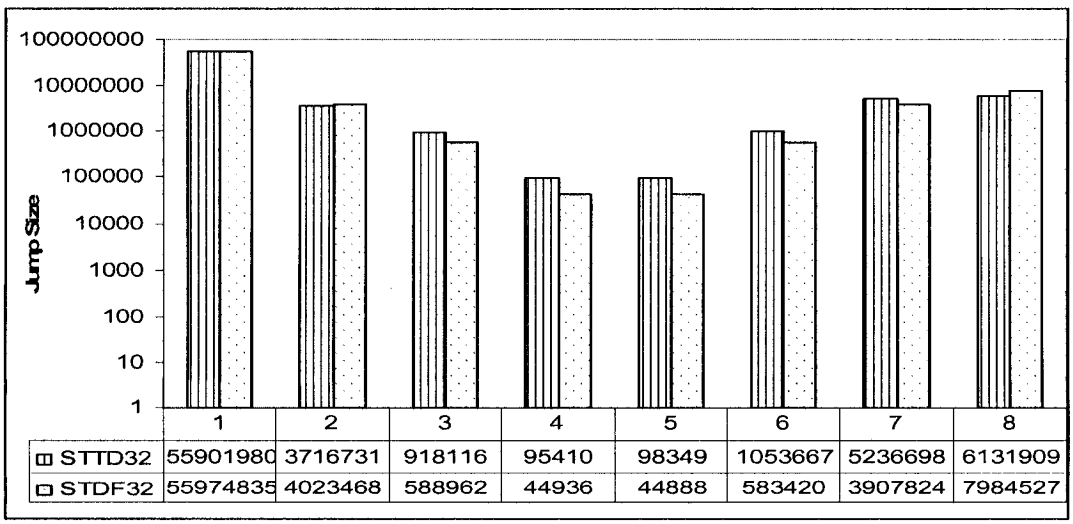
Figure 30 EMS Time for Large Alphabet Sequences

Disk Access Patterns – STTD32 vs. STDF32

While the exact match search times for STTD32 are faster compared to STDF32, it would be useful to study the disk access patterns of both these techniques in order to get a better understanding on the factors that influence their search performance. In this

context, we study the locality of reference exhibited by both the representations during the exact match search (EMS).

The accesses to the suffix tree during the EMS are either forward accesses or backward accesses, depending on the location of the next suffix tree element compared to the current location. We refer to accesses in either direction as *jumps*. Figure 31 shows these jump distributions for Chromosome Y, which result from the EMS for patterns P_1=A, P_2=C, P_3=G, and P_4=T. For clarity, the Y axis in the figure is represented in logarithmic scale. The reason for choosing the shortest possible query length is to ensure that the results will be representative for the whole suffix tree.



	1	2	3	4	5	6	7	8
STTD32	55901980	3716731	918116	95410	98349	1053667	5236698	6131909
STDF32	55974835	4023468	588962	44936	44888	583420	3907824	7984527

Figure 31 Jump Distribution for STTD32 and STDF32

Based on the jump size, measured in the number of suffix tree elements, we classified the jumps into 8 bins, as follows: Bin 1: 0–10 elements; Bin 2: 10–10^2 elements; Bin 3: 10^2–10^3 elements; Bin 4 :10^3–10^4 elements; Bin5 : 10^4–10^5; Bin6: 10^5-10^6; Bin7: 10^6-10^7 and Bin8 for distances larger than 10^7 elements. Larger jump sizes result in buffer misses. In Figure 31, we see that the both STTD32 and STDF32 suffer from huge number of jumps. In the case of STDF32 the huge number of jumps in bin 8 mainly occurs when starting from the root we try to trace the query pattern. If a mismatch occurs, the algorithm tries

to trace the query with the next sibling. However, in the case of STDF32 this sibling node is far away from the current node, depending on the height at which the mismatch occurs. This explains the huge jumps in Bin 8 for STDF32, which in turn results in slower search for STDF32.

Once the query is traced completely, the algorithm needs to traverse the tree and gather all the children. During this phase, STDF32 shows better locality of reference compared to STTD32, since all the child nodes are stored adjacent to each other in STDF32. However, during the retrieval phase in STTD32 there are a lot of buffer misses incurred while traversing the suffix tree and also additional temporary information needs to be stored.

In addition to these factors, there is yet another factor that results in jumps and in turn buffer misses for both STDF32 and STTD32: calculating the *depth* values. The *depth*(s) is the number of characters on the path from the root to the parent of s. As explained before in Section 3.1, the leaf (and branch) node stores the *lp* value and not a pointer to the sequence. The index location of a node s in the sequence is computed by $loc(s) = lp(s) - depth(s)$. Calculating this depth value requires explicit traversal of the suffix tree, which result in jumps. In the case of STDF32, these jumps are relatively short compared to STTD32. This might in turn lead to excessive disk I/O. As a result STDF32 has a better locality of reference that STTD32 for most part of the search [Halachev et al., 2005]. This performance gain for STDF32 due to this locality of reference is however overweighed by the poor locality of reference when the queries are matched.

4.5 Summary

In this chapter we have discussed the various design alternatives to the TDD technique that we considered in our initial study. The first technique that we introduced, STTD32, is a disk-based extension of the WOTD technique while maintaining its index representation. We also introduced a new indexing technique STDF32, where the suffix tree nodes are evaluated and stored in a depth-first order. The following important conclusions are obtained from our study

1) The presence of the two bitmap arrays in TDD results in slower construction and search compared to STTD32 and STDF32.
2) We noted, that the STDF32 technique exhibits better locality of reference for the *tree* structure during construction.
3) While the construction time of STDF32 is slower compared to STTD32 for DNA sequences, it is faster compared to both STTD32 and TDD for protein and natural text. This is because the increased alphabet size results in a higher branching factor of the tree, which in turn leads to less additional information that needs to be stored in the depth-first algorithm. This combined with better locality of reference exhibited by STDF32 results in faster construction time when the alphabet size is larger.
4) The STDF32 representation does not improve the search compared to STTD32, but exhibits better locality of reference for EMS.

Based on the results and experience that we have obtained from our study in this chapter, we are confident to propose our new, efficient, scalable and versatile indexing technique for sequence data.

Chapter 5

Proposed Indexing Techniques for Sequence Data

In this chapter we present our two proposed indexing techniques - Suffix Tree Top-Down 64 bit (STTD64) and Suffix Tree Depth-First 64 bit (STDF64) technique, both extended from their corresponding 32 bit techniques, STTD32 and STDF32, respectively. These two techniques overcome the limitations of the 32 bit versions by handling sequences of sizes larger than 1 GB, and also support faster search. We first discuss the design goals of these new indexing techniques. We then present our proposed suffix tree representations and their corresponding construction algorithms.

Using real life data, we then compare the construction time and index sizes of our techniques to other alternate suffix tree based techniques introduced in the last chapter and also with Enhanced Suffix Array (ESA), which has been incorporated in Vmatch, the most efficient commercially available state-of-the-art technique. These results are complemented with a brief discussion of the search performance.

5.1 Proposed Indexing Techniques

Before discussing the details of these new techniques and evaluating their performance, we summarize the design goals that motivated our proposed techniques.

5.1.1 Design Goals

The motivation of our research is to find an efficient and scalable indexing technique for sequence data. Such a technique should be able to support very large sequences that are common, especially in the biological domain. As discussed in Section 4.4, a TDD type of representation that uses additional bitmap arrays results in slower construction and search times compared to techniques such as STTD32 that encode the leaf and right most bits in the *tree* array. Therefore, a desired new technique that we envision should be able to support large sequences without introducing additional data structures. In Section 4.4 we also showed that one of the search bottlenecks is the calculation of *depth* values. Hence, it is important to reduce the buffer misses that occur during the computation of *depth* values, if we need to achieve search performance close to memory-based search algorithms. With these two desired goals in mind, we propose our new suffix tree techniques, STTD64 and STDF64 both of which are extended from their 32 bit techniques.

5.1.2 Index Representations

Since STTD64 is inspired by STTD32, the two indexing schemes share some common

properties. First, in STTD64 the suffix tree nodes are evaluated and recorded in the same top-down manner as in STTD32. Second, the pointer value associated with each branch node u in STTD64 points to the *firstchild* of u. Finally, in both suffix tree representations, we store the leaf and the rightmost bits for every suffix tree node.

Similarly STDF64 is extended from STDF32. In both the 32 and 64 bit depth-first techniques, the suffix tree nodes are recorded depth-first. The pointer value for each branch node u in STTDF64 points to its first *rightsibling*.

The 64 bit indexing schemes however differ from their 32 bit counterparts in several ways. First, every node in STTD64 and STDF64 occupies a single array element, regardless of being a branch node or a leaf node. Second, each node in both versions of our 32 bit representations is 4 bytes, whereas it is 8 bytes in STTD64/STDF64. Figure 32 depicts the node structures in STTD64/STDF64, respectively.

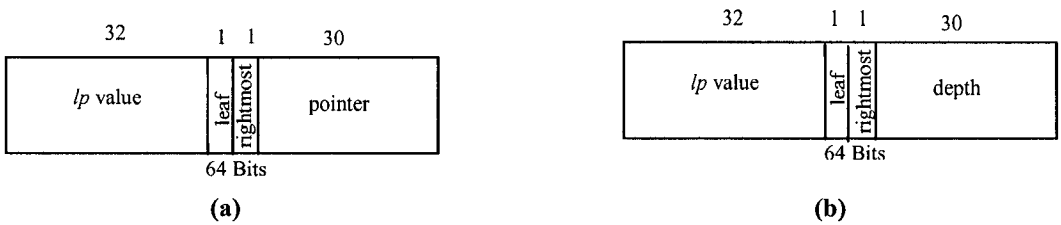


Figure 32 64 Bit node structures (a) Branch node (b) Leaf node

For both the branch and leaf nodes, the first 32 bits store the *lp* value, thus overcoming the 1 GB sequence size limit of the 32 bit techniques. Bit 33 records the leaf bit value and bit 34 records the rightmost bit value. The last 30 bits available for a branch node are used in our STTD64 representation to store the pointer to the first child, while in STDF64 these bits are used to store a pointer to the first right sibling. The third and most important difference between 32 bit and 64 bit techniques is that in the latter we use the last 30 bits available for a leaf node to store its *depth*. The depth of a leaf node s is defined as the

number of characters on the path from the root of the suffix tree to the parent of *s*. For example, for the leaf node 12 in Figure 33(a), the depth is 3 = |GAG|.

For the input sequence *D*=AGAGAGCTT$, Figure 33 shows the corresponding suffix tree graphical illustration and the STTD64 representation. The numbers in the square in Figure 33(a) corresponds to the order in which the nodes are evaluated and created. The first row in Figure 33(b) represents the index location of the suffix tree array. The 32 bit *lp* values are shown in the middle row, and the *pointer/depth* values are shown in the last row. The same convention as for STTD32 is used to mark a leaf node, a rightmost node, and branch node pointers.

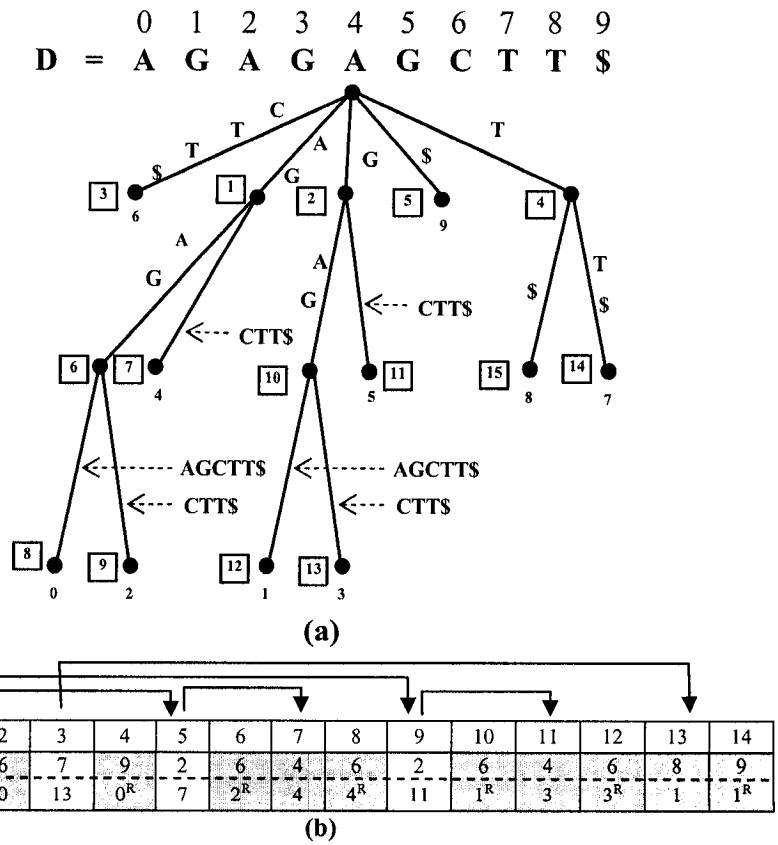
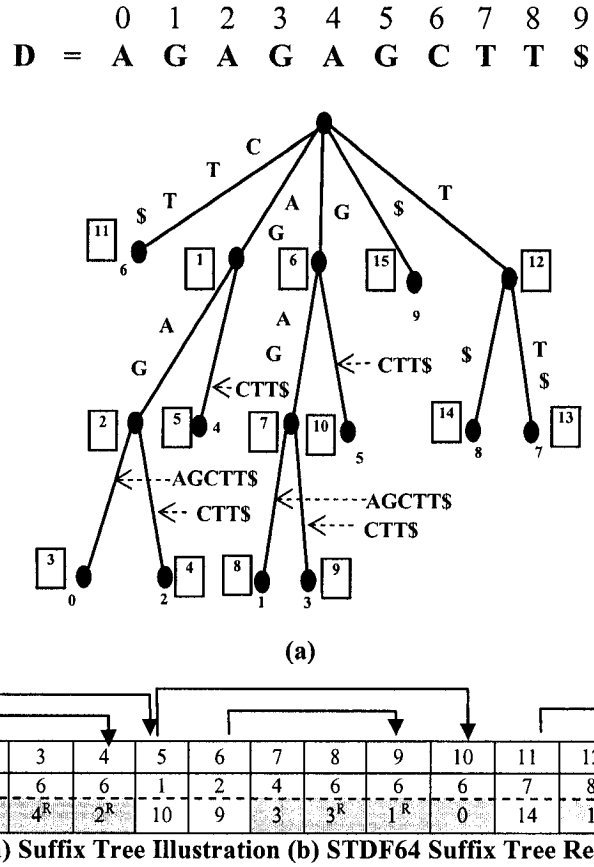


Figure 33 (a) Suffix Tree Graphical Illustration (b) STTD64 Representation of Suffix Tree

For the same input sequence *D*, Figure 34(a) shows the order in which the suffix tree nodes are created in the depth-first order and Figure 34(b) shows the corresponding

STDF64 representation. The same convention as for STTD64 is used for the pointer/depth values and leaf/rightmost nodes.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	2	4	6	6	1	2	4	6	6	6	7	8	9	9
5	4	4	4^R	2^R	10	9	3	3^R	1^R	0	14	1	1^R	0^R

Figure 34 (a) Suffix Tree Illustration (b) STDF64 Suffix Tree Representation

The improvement of the 64 bit techniques over 32 bit techniques comes at a price: increased storage space. Considering various types of input data, our experiments results reported in Section 5.2 indicate that 64 bit techniques on average requires about 13 bytes per character in the input sequence data *D*, thus making it comparable with ESA.

5.1.3 Index Construction

The STTD64 and STDF64 construction algorithms are extensions of their corresponding 32 bit algorithms. Similar to the 32 bit algorithms, the suffixes are partitioned in phase 1 based on the first *prefixlen* symbols. In the second phase, the top-down/depth-first tree is

constructed. STTD64 and STDF64 perform an additional step, Phase 3, where the depth values are recorded. The 64 bit construction algorithm is shown in Figure 35.

ST Construction Algorithms (sequence *D, prefixlen*)
Perform *Phase1* and *Phase2* of STTD32/STDF32
Phase 3: [Depth Filling]
12. Traverse the *Tree* by following the branch node pointers and
    compute and record the *depth* value at each leaf node
end STTD64/STDF64

Figure 35 STDF64/STDF64 Construction Algorithm

5.1.4 Analysis

The construction algorithms for STTD64 and STDF64 have the same worst case time complexity $O(n^2)$ as their 32 bit counterparts. The additional depth filling phase in the 64 bit versions requires an additional $O(n)$ time. These depth values which are stored in the suffix tree, result in an additional $4n$ bytes/character for the 64 bit techniques compared to the 32 bit techniques. Once the 64 bit suffix trees are constructed, all z occurrences of a given pattern P can be found in time $O(z+|P|)$. Since we store the depth values in the leaf nodes of the 64-bit tree, it requires an additional $4n$ disk space compared to the corresponding 32 bit techniques.

5.2 Performance Evaluation

In order to evaluate the performance of our new proposed techniques we compare them to the suffix tree based techniques STTD32 and TDD based on their index creation time, index size, disk access pattern, and exact match search performance. Since STDF32 has a similar construction time, same storage requirement, and slower search performance compared to STTD32 we do not include it in our construction and search experiments.

STDF32 is used only for disk access pattern comparisons. In addition to the suffix tree based techniques we also compare the performance of our 64 bit techniques with the most efficient suffix array based technique, Enhanced Suffix Array (ESA). The system configuration used for these set of experiments is same as that in Section 4.4.

Programs All our programs were written in C and compiled using *gcc* with all optimizations (-O3) enabled. The TDD source code is written in C++ and was obtained from the author's [Tata et al., 2004] website. For ESA based technique, we used Vmatch, a commercially available tool, which implements the ESA index construction and search algorithms. The Vmatch executables were obtained from the author [Vmatch, 2006].

Experimental Datasets To study scalability of the various indexing techniques, we classify our experiments based on the input sequence size into: (1) short sequences of up to 250MB, (2) medium size sequences of up to 1GB, and (3) long sequences with more than 1GB symbols. Let us call them as type 1, type 2, and type 3 sequences, respectively. The basis for this classification is the restriction on the size of the input sequence that could be handled by the considered techniques.

5.2.1 Type 1 Sequences (Up to 250MB)

The 250MB boundary was derived as it is the practical limit for ESA in our machine. We consider the following sequences: (1) all 24 human chromosomes (DNA data, [GenBank, 2005]); (2) the entire Swiss-Prot database (protein data, [SProt, 2005]); and (3) natural texts from the Gutenberg project [Guten, 2005]. The DNA, Protein and Natural text datasets were preprocessed as explained in Section 4.4. Figure 36 shows the sizes of the

chromosomes. The Swiss-Prot database, called *sprot*, is about 75 million characters over an alphabet of 23 symbols and the natural text sample, called *guten80*, contains the first 80 million characters over an alphabet of 95 symbols taken from [28].

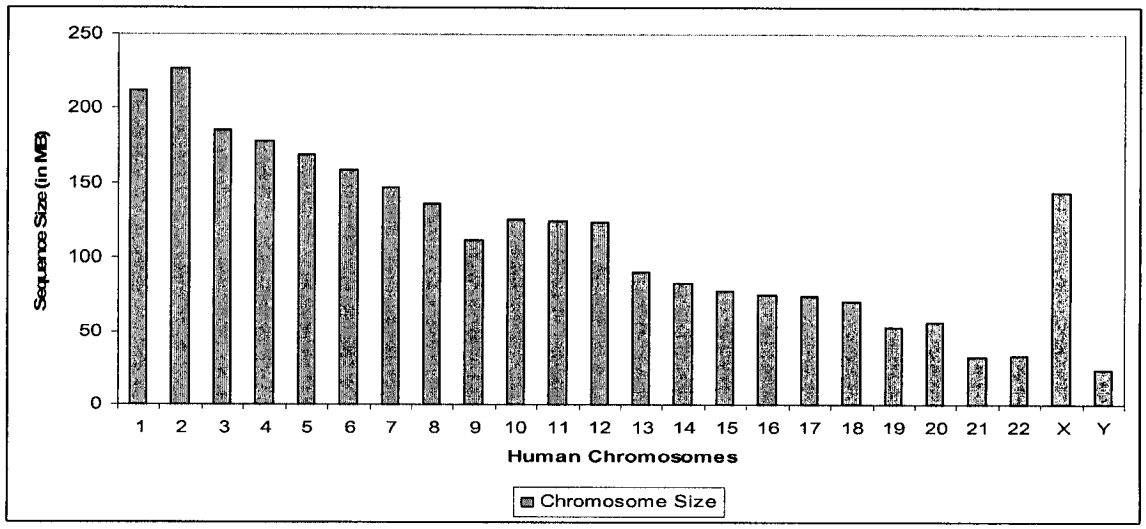


Figure 36 Human Chromosome Sizes

The index construction algorithms that are applicable for type 1 sequences are ESA, TDD, STTD32, and our proposed techniques, STTD64 and STDF64. Figure 37 shows the construction time for DNA data and Figure 38 shows this time for protein and natural text data. It can be seen that ESA has the fastest index construction time for type 1 sequences. This is explained by the fact that ESA is a memory-based technique, which uses no buffering and hence has no buffer overhead. STTD32 is faster compared to TDD due to its better utilization of available memory (STTD32 does not use bitmaps) and smaller index size. The additional filling phase for our 64 bit techniques and the increased index size compared to TDD, result in slower but reasonable construction times for the 64 bit techniques. However, for the 64 bit techniques, STDF64 is faster than STTD64 (unlike in 32 where the depth-first is slower). This is because the *depth* filling phase is similar to EMS where all the leaf nodes are accessed and the *depth* values are

computed. In such cases STDF64 exhibits better locality of reference compared to STTD64. This has been explained in Section 4.4.2.

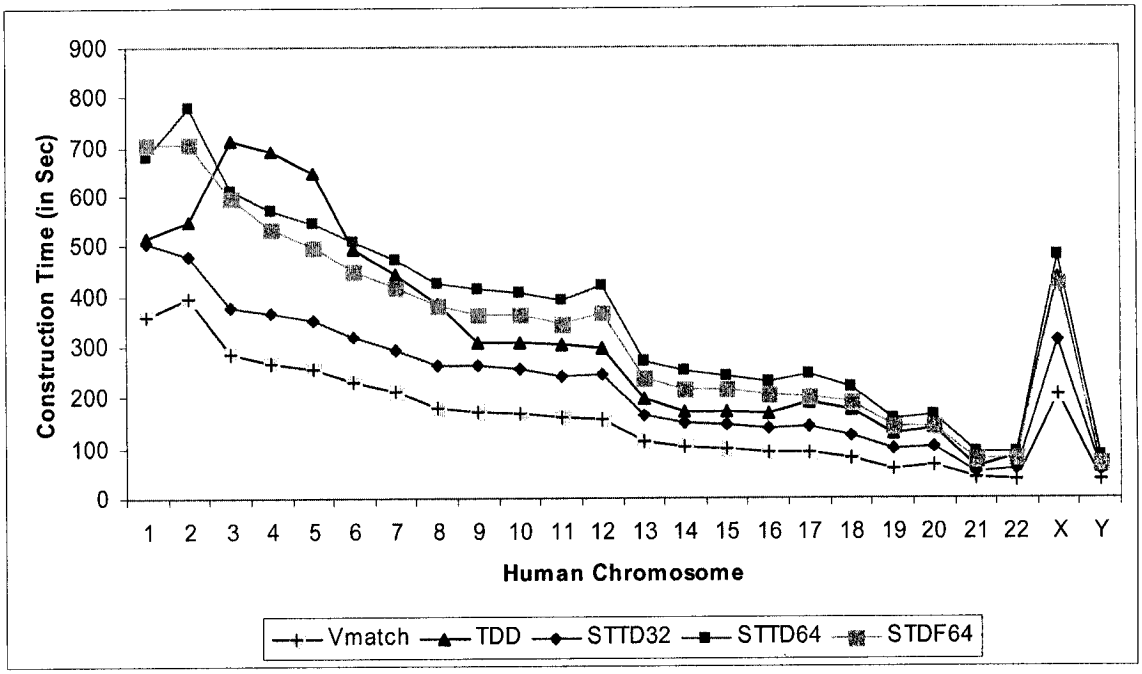
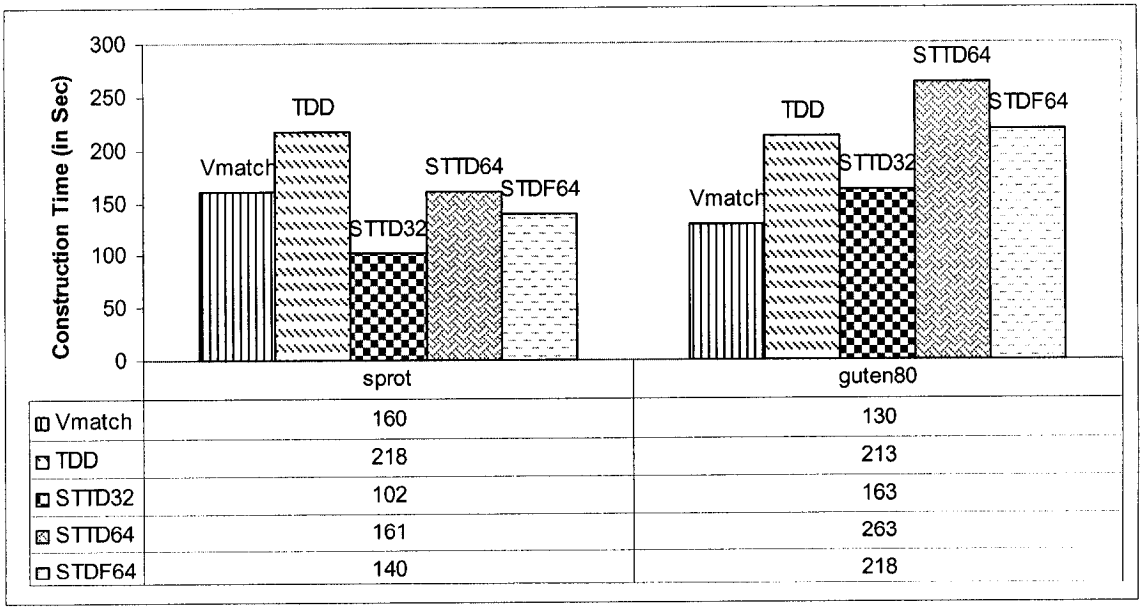


Figure 37 Suffix Tree Index Construction Time for DNA Sequences



	sprot	guten80
Vmatch	160	130
TDD	218	213
STTD32	102	163
STTD64	161	263
STDF64	140	218

Figure 38 Suffix Tree Index Construction for *sprot* and *guten80*

Index Sizes

In terms of storage requirements, for DNA data, STTD32 requires $9.5n$ bytes per

character on average, while TDD, STTD64, and ESA on average require 12.7n, 13.4n, and 13.1n, respectively (Figure 39). For suffix tree based approaches, a larger alphabet size leads to larger branching factor, and hence smaller number of branching nodes. This results in reduced storage requirements. The storage requirements for *sprot* ($|\Sigma|$ = 23) using STTD32, TDD, and STTD64 are 8.5n, 11.8n, and 12.5n, and for *guten80* ($|\Sigma|$ = 95) they are 8.1n, 11.4n, 12.1n, respectively (Figure 40). The ESA storage requirements for *sprot* and *guten80* are 13.3n and 12.2n, respectively.

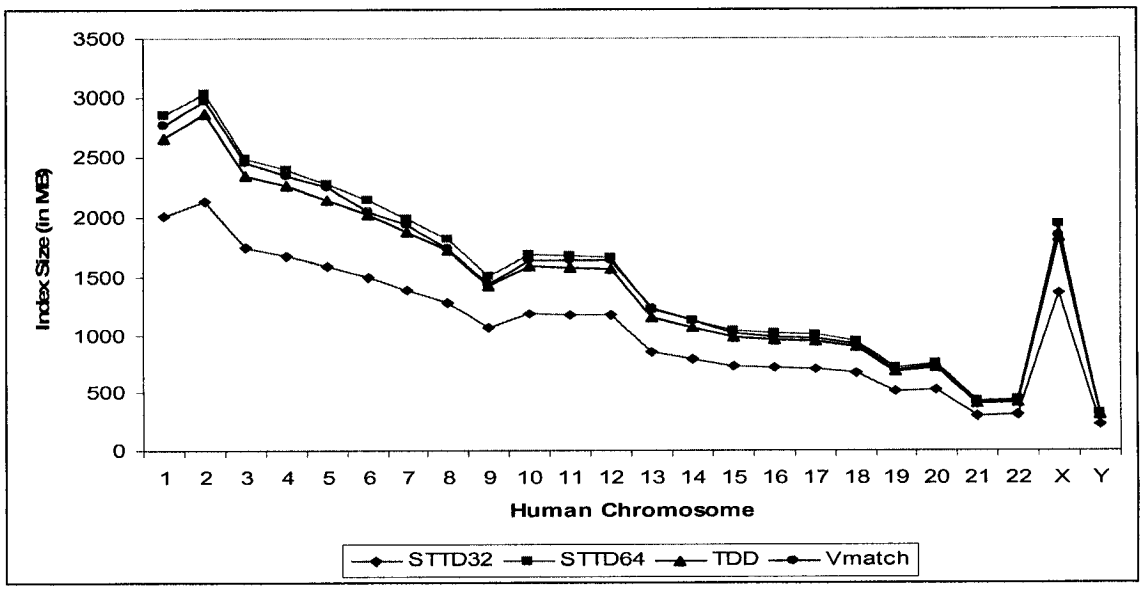


Figure 39 Index Sizes of all Human Chromosomes

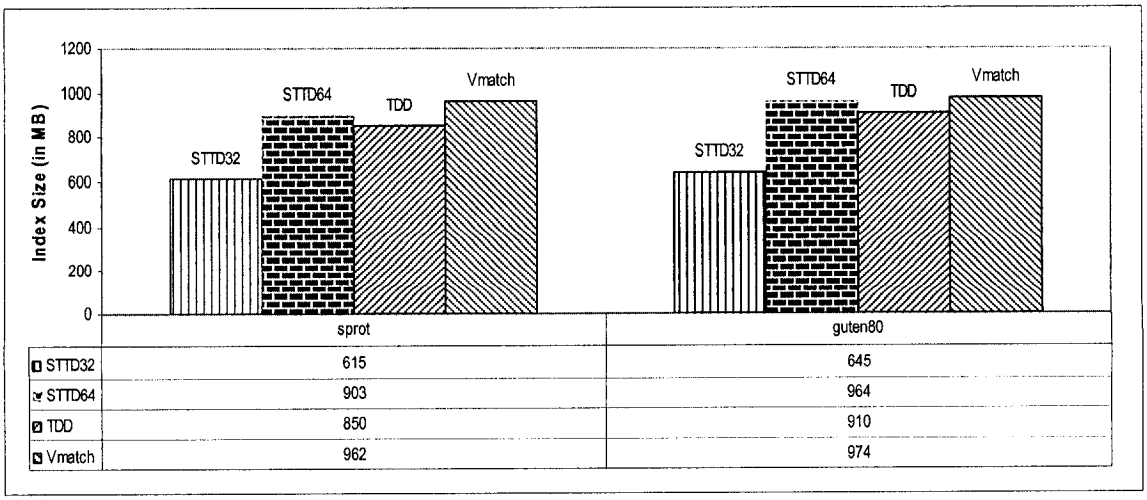


Figure 40 Index Sizes for *sprot* and *guten80*

Exact Match Search Performance

We now discuss the results of our search experiments for the considered techniques. The query sets 7_1200q, 11_1200q, and 15_1200q where obtained by sampling chromosome 2, as already explained in Section 4.4. The query set for *sprot* and *guten80* were also obtained in a similar way as explained in Section 4.4.

The search times for different indexing techniques we used for DNA sequence using query set 11_1200q is shown in Figure 41. Search times using 7_1200q and 15_1200q are similar and hence not shown. The times for protein and natural language text is shown in Figure 42.

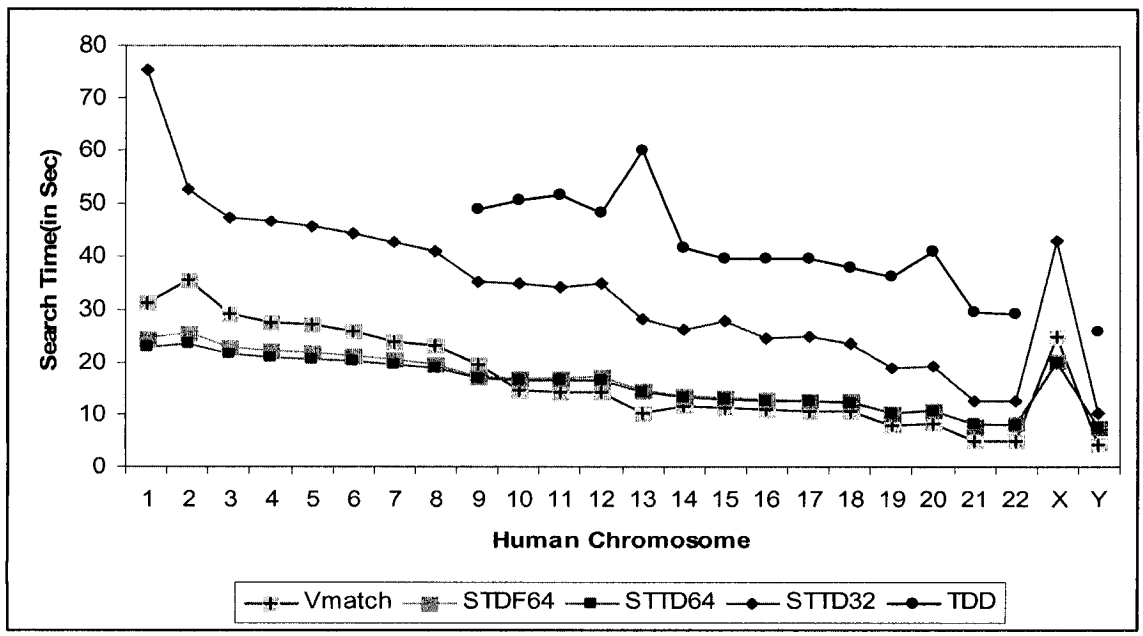
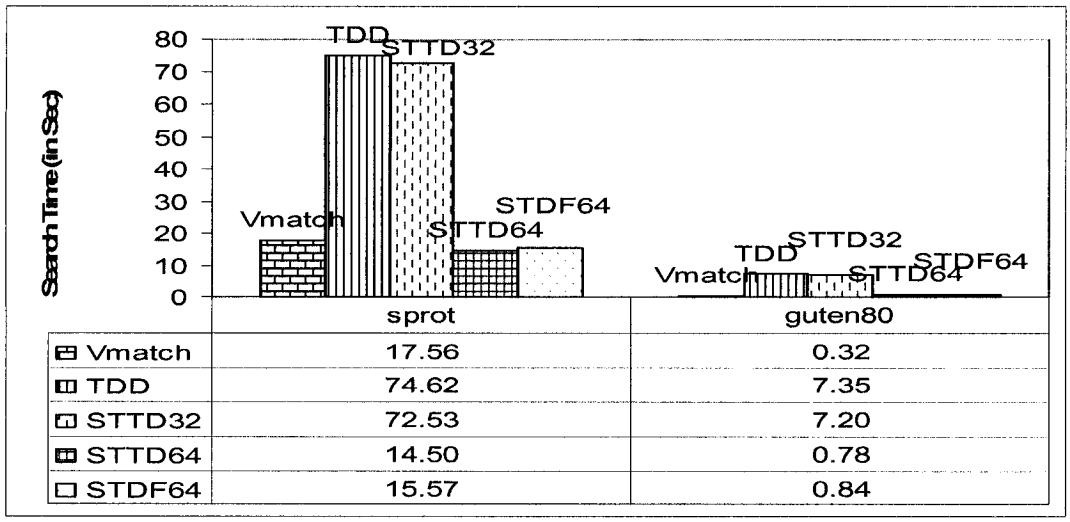


Figure 41 Type 1 DNA sequences Search Times



	sprot	guten80
Vmatch	17.56	0.32
TDD	74.62	7.35
STTD32	72.53	7.20
STTD64	14.50	0.78
STDF64	15.57	0.84

Figure 42 Type1 sprot and guten80 Search Times

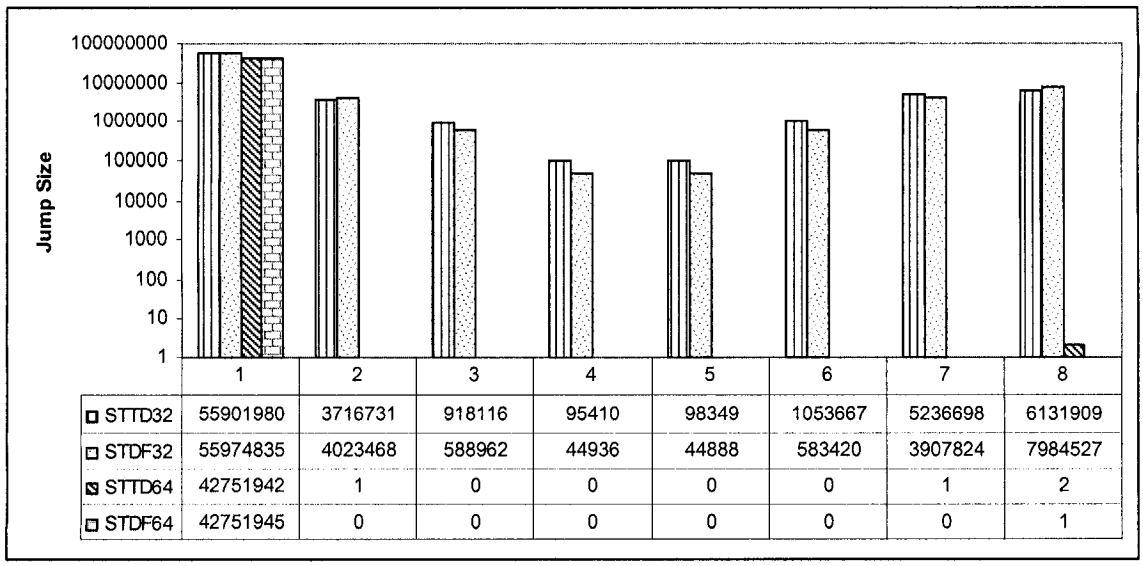
The search performances of STTD64, STDF64 and ESA are comparable (Figures 41 and 42), which is an important result, since the first two are disk-based techniques, while ESA is memory-based. This result validates our idea of storing and using the *depth* value in the 64 bit techniques, which leads to mostly sequential, bulk reads from the disk during the search. The benefit of the *depth* value is further illuminated when comparing STTD64 and STDF64 to STTD32, where the 64 bit techniques are 1.5 times faster on average than STTD32 on DNA data. The superiority of our proposed 64 bit techniques and ESA over STTD32 and TDD becomes even clearer when considering protein data and natural texts, on which we observe an order of magnitude speedup (Figure 42).

TDD was not able to search in larger chromosomes (e.g., chr 1 to 8 and chr X), because the exact match search algorithm provided with TDD is memory based, and for larger chromosomes the available 2GB memory was not sufficient.

Disk Access Patterns

The fast search of the 64 bit algorithms, compared to their 32 bit counterparts can be

better explained by the disk access patterns. This is shown in Figure 43. For clarity, the Y-axis is represented in logarithmic scale. Depending on the jump size, measured by the number of suffix tree elements, we classify the jumps into one of the 8 bins, similar to our previous experiment in Section 4.4.2. It can be seen that storing the depth values in the leaf nodes eliminates the huge jumps that occurs in 32 bit techniques. This in turn translates to faster search. The 64 bit techniques consist of a huge number of jumps in bin 1, which signify mostly sequential reads during search. This in turn leads to better locality of reference and reduced buffer misses.



	1	2	3	4	5	6	7	8
STTD32	55901980	3716731	918116	95410	98349	1053667	5236698	6131909
STDF32	55974835	4023468	588962	44936	44888	583420	3907824	7984527
STTD64	42751942	1	0	0	0	0	1	2
STDF64	42751945	0	0	0	0	0	0	1

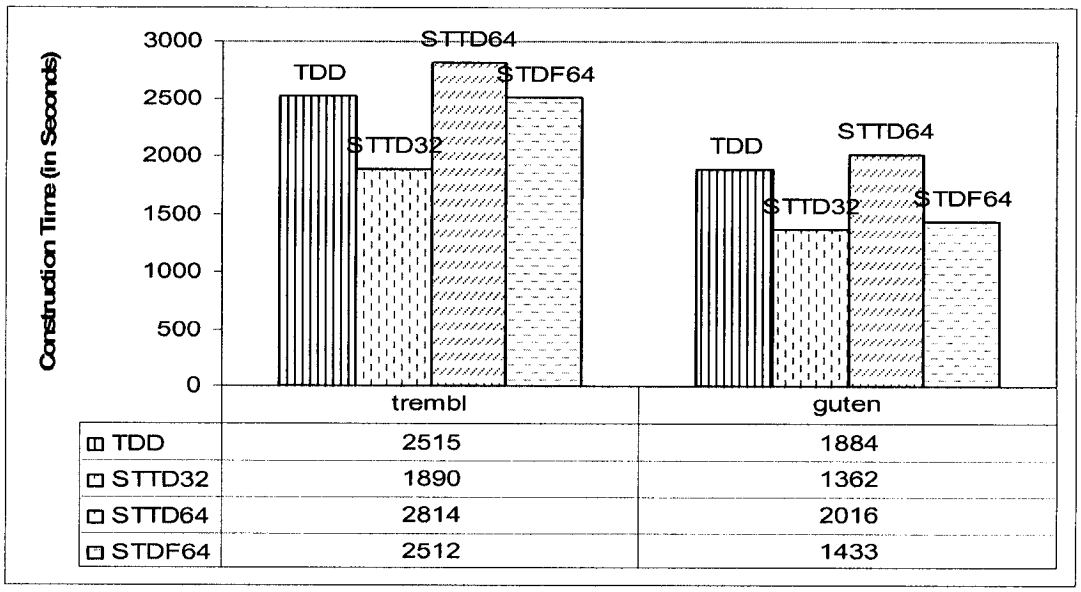
Figure 43 Jump Distributions of 32 bit and 64 bit Techniques

5.2.2 Type 2 Sequences (Up to 1GB)

Type 2 sequences are of size between 250 million and 1 billion symbols. The upper bound is the theoretical limit of the STTD32 representation, as described in Section 3.1. In this set of experiments, we consider a concatenated sequence of chromosome 1 and 2 (DNA data), entire Trembl database (protein data, [ExPASy, 2005]), and the entire

Gutenberg collection [Guten, 2005]. We preprocessed the data as explained before to obtain *chr1_2* sequences of about 440 million symbols, *trembl* sequence of about 840 million symbols, and the *guten* sequence of about 400 million symbols. The indexing techniques that are applicable for this input size range include STTD32, STTD64, STDF64 and TDD.

Figure 44 shows the construction times for sequences with large alphabet size i.e., *trembl* and *guten*. These results are consistent with the corresponding results for type 1 sequences.



	trembl	guten
TDD	2515	1884
STTD32	1890	1362
STTD64	2814	2016
STDF64	2512	1433

Figure 44 Suffix Tree Construction Times for Trembl and Guten

The construction times for the DNA sequence *chr1_2* using STTD32, STTD64 and STDF64 techniques are 19min, 30min, and 28min, respectively. The inefficiency of TDD bitmap arrays was further illuminated when we tried to construct the suffix tree for this DNA sequence. For TDD, we stopped the algorithm after an hour and a half, and only a very small part of the TDD suffix tree was completed for *chr1_2*. This is because when the alphabet size is small and the size of the sequence is large, the partitions are large.

Hence, the *suffixes* and *temp* partitions also need to be buffered in addition to the *tree* as a large chunk of the memory is allocated for the two bitmap arrays. This results in poor performance, as observed in our experiment.

The storage requirements for TDD, STTD32, and STTD64 for type 2 protein and natural text sequences are shown in Figure 45. STDF64 has the same space requirements as STTD64 and hence not shown in the figure. While the index sizes for STTD32 and STTD64 are consistent with those obtained for short sequences, the size of the TDD index for *guten* sequence was found to be 25.5*n*, which was not expected. This is probably due to the heuristics they used for allocating space to the bitmap arrays before the actual construction of the tree.

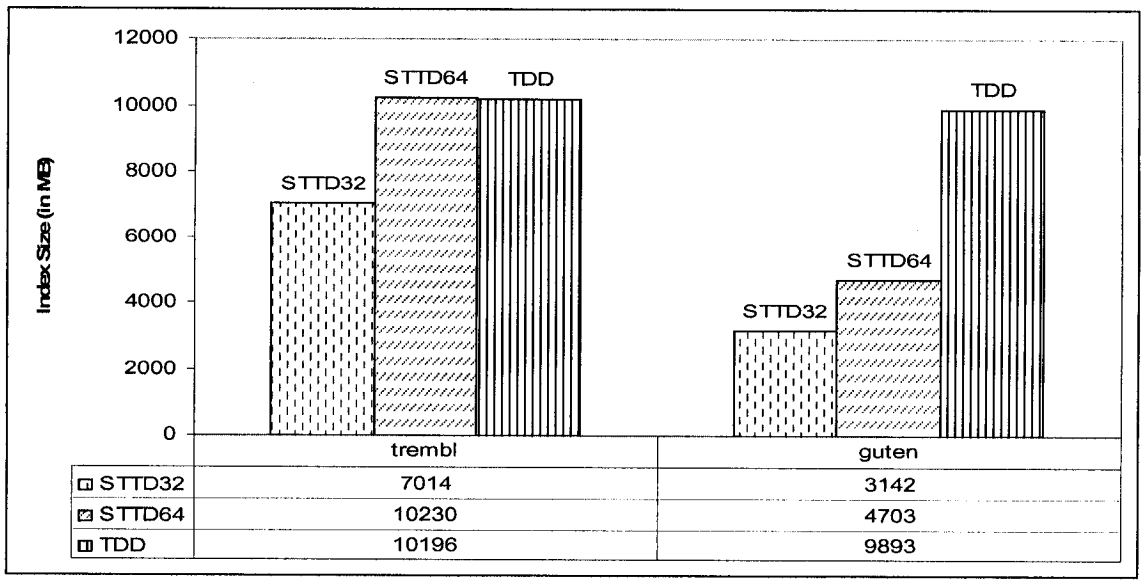


Figure 45 Index Size for Type 2 Sequences

The measured index sizes for chr1_2 for STTD32 and STTD64/STDF64 techniques are 4GB and 5.8GB, respectively.

Search Performance

The measured search times for the considered indexing techniques for sequences *chr1_2*, *trembl*, and *guten* are consistent with those observed for type1 sequences and hence not shown here.

5.2.3 Type 3 Sequences (Up to 4GB)

The largest input sequence that we consider is the entire human genome obtained by concatenating the preprocessed 24 human chromosomes of size 2.8 GB. This sequence is compressed by encoding each character as a 4 bit binary value as done in [Tata et al., 2004]. This compression process enables us maintain the whole sequence in memory during the index construction process. The only applicable techniques for type 3 are TDD, STDF64, and STTD64. The TDD code available to us does not support indexing sequences of such size. For comparison purposes, we use the TDD construction time reported in [Tata et al., 2004] which is 30 hours. We noted that for type 1 and 2 sequences, running TDD on our machine results in faster construction times compared to those reported in [Tata et al., 2004], possibly due to hardware differences. Using our STTD64 and STDF64 techniques, we could construct the suffix tree for then entire human genome in approximately 16 hours and they outperform TDD (even when accounting for hardware differences). This superiority of our techniques is due to better utilization of the available memory, by using dynamic buffering and avoiding leaf and rightmost bit arrays.

The observed STTD64 and STDF64 index size of the human genome is around 36GB (12.8 bytes per character). The index size obtained using TDD is not reported in the literature and hence a comparison based on the space requirements with our techniques is not possible. Theoretically, TDD should have a smaller index size compared to our techniques, however, their heuristics for allocating the bitmap arrays results in very large bitmap arrays in some cases (e.g. index size for the entire Gutenberg sequence was found to be 25.5n). Hence, we are not conclusive in our comparison with TDD on the index size.

5.3 Summary

In this chapter we introduced our proposed 64 bit techniques, STTD64 and STDF64 for indexing sequence data. The 64 bit techniques extend from their 32 bit counterparts and overcome their limitations. In terms of representation, branch and leaf nodes in both the 64 bit techniques occupy a suffix tree element, which is of size 8 bytes. This allows the 64 bit techniques to overcome the 1GB input sequence size limitation of the 32 bit techniques. Our proposed 64 bit techniques can handle sequences as large 4GB, similar to the best known suffix tree based technique TDD. Another difference between the 64 bit and 32 bit techniques is that, in the former we store the *depth* value in the leaf nodes, which improves the locality of reference during search applications. While this results in improved search performance, it results in a larger index size (4n bytes more than the 32 bit tree).

Through our experiments using real life experiments (in Section 5.2) we show that even

though our STTD64 and STDF64 techniques have a higher construction time for type 1 sequences, due to the increased index size and depth filling phase, they outperform the alternative suffix tree based techniques like TDD, STTD32 and STDF32. The search performance of our 64 bit, disk-based techniques is comparable to ESA, an in-memory solution which uses suffix array. The results of our search experiments justify our representation model and the idea of storing the depth values in the suffix tree index nodes. For type 2 sequences, the only comparable techniques to our proposed 64 bit techniques are TDD and STTD32. For the 32 bit and 64 techniques, our experiments show that their performance is similar to type 1 sequences. However, for TDD, the performance bottleneck due to the presence of bitmap arrays is more visible for type 2 sequences. This results in slower construction times. For type 3 sequences, we found that both STTD64 and STDF64 outperform TDD, the best known disk-based suffix tree indexing technique.

Based on the experimental results for all the 3 types of sequences, we conclude that our new 64 bit indexing techniques STTD64 and STDF64 provide the best tradeoff between scalability, index size, and search performance compared to other suffix tree and suffix array based solutions. Our partial results for *k*-mismatch searching[1], indicate that our 64 bit techniques are more promising, making them a desired indexing technique for incorporating in biological tools.

[1] The manuscript of this study is currently being prepared for submission to a journal.

Chapter 6

Conclusion and Future Work

Explosive growth of sequence data in the form of digital texts, time series, biological databases, etc., has resulted in an urgent need for new and scalable data access techniques. In case of biological sequences, this is a challenging task because biological data do not have any structure and hence cannot be broken into meaningful words. In this context, we proposed new suffix tree based indexing techniques for sequence data, and compared their performance with existing techniques on the basis of construction time, space requirement, and search efficiency.

We studied existing techniques for indexing sequence data and focused on suffix tree based techniques due to their efficient support for numerous search applications. We then considered two suffix tree based techniques, WOTD [Giegerich et al., 2003] which is the most space efficient technique and TDD [Tata et al., 2004] the only suffix tree based techniques that could build suffix trees for sequences as large as the entire human genome. Based on the initial evaluation of these two techniques, we wanted to explore possible extensions to the WOTD technique. This resulted in our 32 bit disk-based techniques STTD32 and STDF32 [Halachev et al., 2005], which could support sequences as large as 1 billion symbols. While both the 32 bit techniques have similar construction time and space requirements, they outperform TDD based on all our comparison criteria. Two important observations that we made in this study was STDF32 had a better locality

of reference than STTD32 for most part of the search and the calculation of *depth* resulted in buffer misses.

While our 32 bit techniques perform better than TDD, they do not handle sequences bigger than 1 billion. In order to overcome this limitation in our 32 bit techniques, we proposed and developed our two new indexing techniques STTD64 [Halachev et al., 2007] and STDF64. Both these techniques have the same storage requirement, similar construction, and exact match search times. While our 64 bit techniques have a slower construction time and bigger index size compared to TDD, they perform faster exact match searches. Comparing our techniques with ESA [Abouelhoda et al., 2002], a state-of-the-art memory-based suffix array technique, we show that our 64 bit disk-based techniques provided comparable search times and have similar storage requirements for sequences that ESA can handle. While ESA, being a memory-based technique has a faster construction time for sequences it can support (up to 250 million symbols) compared to our 64 bit techniques, for longer sequences our 64 bit techniques are the only choice.

Our experimental results show that both STTD64 and STDF64 have similar performance. However, our research objective would be to use one of these 64 bit techniques for a variety of search applications. This would require further experimentation, with other types of search algorithm. This is currently in progress.

Table 1 shows the various types of sequences and the indexing techniques that are applicable for each type. We classified the sequences into 3 types: (1) short sequences (up to 250MB), (2) medium Sequences (up to 1GB), and (3) large sequences (up to 4GB).

For each sequence size type, the table indicates the applicable techniques, using a check mark (√).

Sequence Size		Short < 250MB	Medium 250MB – 1GB	Large > 1GB
Techniques	ESA	√	NA	NA
	TDD	√	√	√
	STTD32	√	√	NA
	STDF32	√	√	NA
	STTD64	√	√	√
	STDF64	√	√	√

Table 1 Applicable Technique for each Sequence Type

Table 2 summarizes our findings in this study and shows the recommended indexing technique for each sequence type based on the three evaluation parameters – index size, construction time and exact match search time. The shaded regions in the table indicate that our proposed techniques are the best for the considered parameter for the particular sequence type. We did not find any report on the TDD index size for the entire human genome (shown as *?*) and thus no comparison is recorded in the table for index size for type 3 sequences.

Sequence Size		Short < 250MB	Medium 250MB – 1GB	Large > 1GB
Parameters	Index Size	STTD32/STDF32	STTD32/STDF32	?
	Construction	ESA	STTD32	STTD64/STDF64
	Search	ESA or STTD64/STDF64	STTD64/STDF64	STTD64/STDF64

Table 2 Ranking of Indexing Techniques

As a future work, we would like to develop balanced partitioning schemes for our suffix tree construction algorithms. Balanced partitions would enable us to handle most of the *suffixes* and *temp* data structure in memory, resulting in faster construction times. Another option would be to partition the sequence into *l*-disjoint subsets (rather than partitioning the suffixes), similar to [Tian et al., 2005] and build a suffix tree for each

subset. While this technique would help reduce the random access to the sequence, it would require efficient suffix tree merging applications. This would be worth investigating, especially when the size of the sequences is more than 4GB, which is the memory address limit in 32 bit machines. Also, the suffix tree construction algorithm fits perfectly the parallel execution paradigm. This is because once the *suffixes* are partitioned, each subtree can be built independently on multiple processors at the same time. This would be another possible future work.

The *depth* value stored in the leaf nodes of STTD64/STDF64 representations is similar to the *lcp* value in augmented suffix array [Manber and Myers, 1991]. As the computation of the *lcp* values between all pairs of lexicographically subsequent suffixes is important for many approximate matching solutions [Gusfield, 1997], the presence of *depth* value in our representation might be of potential advantage. We believe it would result in more efficient approximate matching (k-mismatch search, k-difference search), which we are currently investigating as a part of our ongoing research work.

Bibliography

[Abouelhoda et al., 2002] Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. *The Enhanced Suffix Array and its Applications to Genome Analysis.* In Proc. 2nd Intl. Workshop on Algorithms in Bioinformatics, Springer-Verlag, Lecture Notes in Computer Science, Vol. 2452, Pages 449-463, 2002.

[Abouelhoda et al., 2004] Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. *Replacing Suffix Trees with Enhanced Suffix Arrays.* In Journal of Discrete Algorithms, Vol. 2(1), Pages 53-86, 2004.

[Altschul et al., 1990] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., and Lipman, D.J. *Basic Local Alignment Search Tool.* In Journal of Molecular Biology, 215, Pages 403-410, 1990.

[Altschul et al., 1997] Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D.J. *Gapped BLAST and PSI-BLAST: A New Generation of Protein Database Search Programs.* In Nucleic Acids Research, Vol. 25, Pages 3389-3402, 1997.

[Andersson and Nilsson, 1995] Andersson, A. and Nilsson, S. *Efficient Implementation of Suffix Trees. In Software-Practice and Experience*, Vol. 25(2), Pages 129-141, 1995.

[Bedathur and Haritsa, 2004] Bedathur, S.J. and Haritsa, J.R. *Engineering a Fast Online Persistent Suffix Tree Construction.* In Proc. 20th Intl. Conference on Data Engineering (ICDE), Pages 720-731, 2004.

[Bieganski et al., 1994] Bieganski, P., Riedl, J., Carlis, J.V and Retzel, E.F. *Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation*. In Proc. 27th Annual Hawaii Intl. Conference on System Sciences, Vol. 5, Pages 35-44, 1994.

[Bieganski, 1995] Bieganski, P. *Genetic Sequence Data Retrieval and Manipulation Based on Generalized Suffix Trees*, PhD thesis, University of Minnesota, USA, 1995.

[Burkhardt et al., 1999] Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H.P., Rivals E. and Vingron M. *q-Gram Based Database Searching Using a Suffix Array*. In Proc. 3rd Annual Intl. Conference on Computational Molecular Biology (RECOMB99), Pages 77-83, 1999.

[Burkhardt and Kärkkäinen, 2003] Burkhardt, S. and Kärkkäinen, J. *Fast Lightweight Suffix Array Construction and Checking*. In Proc. 14th Annual Symposium on Combinatorial Pattern Matching, LNCS 2676, Springler-Verlag, Pages 55-69, 2003.

[Crauser and Ferragina, 2002] Crauser, A. and Ferragina, P. *A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory*. In Algorithmica, Vol. 32, Pages 1-35, 2002.

[Delcher et al., 1999] Delcher, A.L., Kasif, S., Fleischmann, R.D., Peterson, J., White, O., and Salzberg, S.L. *Alignment of Whole Genomes*. In Nucleic Acids Research, Vol. 27(11), Pages 2369-2376, 1999.

[Dementiev et al., 2005] Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. *Better External Memory Suffix Array Construction*. In Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX '05), SIAM, 2005.

[DDBJ, 2005] *DNA Data Bank of Japan (DDBJ)*. http://www.ddbj.nig.ac.jp/, Last accessed, 2005.

[ExPASy, 2005] *Expert Protein Analysis System (ExPASy) Proteomics Server.* http://us.expasy.org, Last accessed, 2005.

[EMBL, 2005] *European Molecular Biology Laboratory (EMBL).* http://www.ebi.ac.uk/embl/, Last accessed, 2005.

[Farach et al., 1998] Farach, M., Ferragina, P., and Muthukrishnan, S. *Overcoming the Memory Bottleneck in Suffix Tree Construction.* In Proc. 39th Annual Symposium on Foundations of Computer Science (FOCS98), 1998.

[Ferragina and Grossi, 1999] Ferragina, P. and Grossi, R. *The String B-tree: A New Data Structure for String Search in External Memory and its Applications.* In Journal of the ACM (JACM), Volume 46(2), Pages 236-280, 1999.

[Fredkin et al., 1960] Fredkin, E., Beranek, B. and Newman Inc. *Trie Memory.* In Communications of the ACM, Vol. 3(9), Pages 490-499, 1960.

[GenBank, 2005] *GenBank.* http://www.ncbi.nlm.nih.gov/Genbank/index.html, Last accessed, 2005.

[Giegerich and Kurtz, 1995] Giegerich, R. and Kurtz, S. *A comparison of Imperative and Purely Functional Suffix Tree Constructions.* In 5th European Symposium on Programming, Pages 187-218, 1994 (Published in 1995).

[Giegerich et al., 2003] Giegerich, R., Kurtz, S., and Stoye, J. *Efficient Implementation of Lazy Suffix Trees.* In Software – Practice and Experience, Vol. 33, Pages 1035-1049, 2003.

[Gusfield, 1997] Gusfield, D. *Algorithms on Strings, Trees and Sequences, Computer Science and Computational Biology.* Cambridge University Press, 1997.

[Guten, 2005] *Project Gutenberg.* http://www.gutenberg.org, Last accessed, 2005.

[Halachev et al., 2005] Halachev, M., Shiri, N., Thamildurai, A. *Exact Match Search in Sequence Data Using Suffix Trees*, In Proc. ACM 14th Intl. Conf. on Information and Knowledge Management (CIKM), Bremen, Pages 123-130, 2005.

[Halachev et al., 2007] Halachev, M., Shiri, N., Thamildurai, A. *Efficient and Scalable Indexing Techniques for Biological Sequence Data.* In Proc. 1st Intl. Conference on Bioinformatics Research and Development (BIRD), LNBI Springer, Berlin, Pages 464-479, 2007.

[Horspool, 1980] Horspool, R.N. *Practical Fast Searching in Strings.* In Software—Practice and Experience, Vol. 10(6), Pages 501–506, 1980.

[Hunt et al., 2000] Hunt, E., Irving, R.W. and Atkinson, M.P. *Persistent Suffix Trees and Suffix Binary Search Trees as DNA sequence Indexes.* Technical Report no. TR-2000-63 of the Computing Science, University of Glasgow, 2000.

[Hunt et al., 2001] Hunt, E., Atkinson, M. P., and Irving, R. W. *A Database Index to Large Biological Sequences.* In Proc. 27th Intl. Conference on Very Large Databases (VLDB), Pages 139-148, 2001.

[Itoh and Tanaka, 1999] Itoh, H. and Tanaka, H. *An Efficient Method for In Memory Construction of Suffix Arrays.* In Proc. IEEE Symposium on String Processing and Information Retrieval, Pages 81-88, 1999.

[Kärkkäinen and Sanders, 2003] Kärkkäinen, J. and Sanders, P. *Simple Linear Work Suffix Array Construction.* In Proc. International Colloquium of Automata, Languages and Programming, in Lecture Notes in Computer Science, Vol. 2719, Pages 943-955, Springer-Verlag, Berlin, 2003.

[Kim et al., 2003] Kim, D.K., Sim, J.S., Park, H., and Park, K., *Linear-time Construction of Suffix Arrays*. In Proc. Annual Symposium on Combinatorial Pattern Matching, in Lecture Notes in Computer Science, Vol. 2676, Pages 186-199, Springer-Verlag, Berlin, 2003.

[Ko and Aluru, 2003] Ko, P. and Aluru, S. *Space Efficient Linear Time Construction of Suffix Arrays*. In Proc. Annual Symposium on Combinatorial Pattern Matching, in Lecture Notes in Computer Science, Vol. 2676, Pages 200-210, Springer-Verlag, Berlin, 2003.

[Kulla and Sanders, 2006] Kulla, F. and Sanders, P. *Scalable Parallel Suffix Array Construction*. In EuroPVM/MPI, 2006.

[Kurtz, 1999] Kurtz, S. *Reducing the Space Requirement of Suffix Trees*. In Software – Practice and Experience, Vol. 29(13), Pages 1149-1171, 1999.

[Kurtz and Schleiermacher, 1999] Kurtz S. and Schleiermacher C. *REPuter: Fast Computation of Maximal Repeats in Complete Genomes*. In Bioinformatics, 1999.

[Lam et al., 2002] Lam, T.W., Sadakane, K., Sung, W.K., and Siu Ming Yiu. *A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays*. In Proc. 8th Annual International Conference on Computing and Combinatorics, LNCS, Springer, Pages 401-410, 2002.

[Manber and Myers, 1991] Manber, U. and Myers, G. *Suffix Arrays: A New Method for On-line String Searches*. In SIAM Journal on Computing, Vol. 22(5), Pages 935-948, 1991.

[Manzini and Ferragina, 2004] Manzini, G. and Ferragina, P. *Engineering a Lightweight Suffix Array Construction Algorithm.* In Algorithmica, Vol. 40(1), Pages 33-50, 2004.

[McCreight, 1976] McCreight, E.M. *A Space-economical Suffix Tree Construction Algorithm.* In Journal of the ACM, Vol. 23(2), Pages 262-272, 1976.

[Miller et al., 1999] Miller, C., Gurd, J., and Brass, A. *A RAPID Algorithm for Sequence Database Comparisons: Application to the Identification of Vector Contamination in the EMBL Databases.* In Bioinformatics, Vol. 15(2), Pages 111-121, 1999.

[Morrison, 1968] Morrison, D. *PATRICIA- Practical Algorithm to Retrieve Information Coded in Alphanumeric.* In Journal of the ACM, Vol. 15(4), Pages 514-534, 1968.

[Neelapala et al., 2004] Neelapala, N., Mittal, R., and Haritsa, J. R. *SPINE: Putting Backbone into String Indexing.* In Proc. 20th International Conference on Data Engineering, 2004.

[Navarro and Baeza-Yates, 1998] Navarro, G. and Baeza-Yates, R. *A Practical q-Gram Index for Text Retrieval Allowing Errors.* In CLEI Electronic Journal, Vol. 1(2), 1998.

[Navarro and Baeza-Yates, 2000] Navarro, G. and Baeza-Yates, R. *A Hybrid Indexing Method for Approximate String Matching.* In Journal of Discrete Algorithms, Vol. 1(1), Pages 205-239, 2000.

[Navarro, 2000] Navarro, G. *A Guided Tour to Approximate String Matching.* In ACM Computing Surveys, Vol. 33(1), Pages 31-88, 2000.

[NCBI, 2007] *NCBI: National Center for Biotechnology Information.* http://www.ncbi.nlm.nih.gov, Last accessed, 2007.

[Puglisi et al., 2005] Puglisi, S.J., Smyth, W.F., and Turpin, A. *A Taxonomy of Suffix Array Construction Algorithms*. In Proc. Prague Stringology Conference, 2005.

[Sadakane and Shibuya, 2001] Sadakane, K. and Shibuya, T. *Indexing Huge Genome Sequences for Solving Various Problems*. In Genome Informatics, Pages 175-183, 2001.

[Sanger, 2006] Sanger Institute. http://www.sanger.ac.uk/Info/Press/2006/060117.shtml, Last accessed, 2007.

[Seward, 2000] Seward, J. *On the performance of the BWT sorting algorithms*. In Proc. Conference on Data Compression, Pages 173-182, 2000.

[Skiena, 1998] Skiena, S. *Who is interested in algorithms and why? Lessons from the Stony Brook Algorithms Repository*. In Proc. 2nd Workshop on Algorithm Engineering, 1998.

[SProt, 2005] *Swiss-Prot Protein Database*. http://ca.expasy.org/sprot/, Last accessed, 2005.

[Tata et al., 2004] Tata, S., Hankins, R.A., and Patel, J.M. *Practical Suffix Tree Construction*. In Proc. 30th VLDB Conference, 2004.

[Tian et al., 2005] Tian, Y., Tata, S., Hankins, R.A., and Patel, J.M. *Practical Methods for Constructing Suffix Trees*. In The VLDB Journal, Vol. 14 (3), Pages 281-299, 2005.

[Ukkonen, 1995] Ukkonen, E. *On-line construction of Suffix Trees*. In Algorithmica, Vol. 14(3), Pages 249-260, 1995.

[Vmatch, 2006] *The Vmatch Large Scale Sequence Analysis Software*. http://www.vmatch.de/, Last accessed, 2006.

[Weiner, 1973] Weiner, P. *Linear Pattern Matching Algorithms*. In Proc. 14th Annual IEEE Symposium on Switching and Automata Theory, 1973.

[Witten et al., 1999] Witten, I.H., Moffat, A., and Bell, T.C. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.