Management of Biological Sequences Using Suffix Trees

MIHAIL HALACHEV

A Thesis

In the Department of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy at

Concordia University

Montreal, Quebec, Canada

May 2009

Your file Votre référence
ISBN: 978-0-494-63445-5
Our file Notre référence
ISBN: 978-0-494-63445-5

ABSTRACT

Management of Biological Sequences Using Suffix Trees

Mihail Halachev, Ph.D.
Concordia University, 2009

The amount of available biological sequences, represented as strings over the DNA and protein alphabets, grows at phenomenal rate. Supporting various search tasks over such data efficiently requires development of sophisticated indexing techniques. Recently, suffix tree (ST) and suffix array (SA) received considerable attention as suitable data structures in this context. However, existing solutions often focus on either efficiency or scalability, but not both. Further, some of the solutions require advanced computational resources or are tailored towards a specific application.

We investigate, both theoretically and experimentally, ways to improve efficiency and scalability in management of biological sequence data. Our goal is to develop an indexing technique that is reasonable in construction time and space utilization, and supports efficiently versatile search applications in biological sequences of various sizes, running on a typical desktop computer.

The contributions of this research include development of a ST based indexing technique, called HST, together with exact and approximate search algorithms that use the index. The results of our experiments indicate that the index construction cost is comparable to other ST based techniques, such as TDD and Trellis, in terms of construction time and main memory requirement. While HST exhibits slower construction time than Vmatch, the best known SA based solution, with the same amount of main memory HST can handle sequences that are an order of magnitude longer. In

terms of the index size, HST is comparable to TDD and Vmatch, which is half of the Trellis index size.

We also develop efficient and scalable search applications using HST, including exact match, k-mismatch, and structured motif search. Our experiments using real-life sequences indicated that for short sequences (e.g., human chromosomes), our exact match search is comparable to Vmatch, about 3 times faster than TDD, and more than 10 times faster than Trellis. Further, HST can be used to search directly in longer DNA sequences, as opposed to partitioning such a sequence and search in the parts – the only option to follow with Vmatch. We found that a direct exact match search using HST is twice faster when searching in the entire human genome, compared to using Vmatch on parts. Compared to Trellis, which can handle direct search in human genome, HST was more than 20 times faster. To further compare performance of HST and Vmatch, we considered k-mismatch search. Our results indicated significant improvement of the HST based solution over Vmatch, ranging from 2 to 9 times faster k-mismatch search on average, for short and long sequences, respectively. For structured motif search, HST was about 6 times faster than SMOTIF1, the best known structured motif search tool.

Acknowledgements

I would like to take the opportunity to thank the many people who supported me during my PhD work and made this thesis possible.

First, I acknowledge the invaluable contribution of my supervisor Dr. Nematollaah Shiri. Throughout my studies he was a constant source of support, motivation, and encouragement. I am grateful for his precious guidance, numerous insightful conversations, and prompt feedback, for which I thank him.

I would like to thank Dr. Christopher Baker and Dr. Gregory Butler for sharing their knowledge and their helpful suggestions.

My colleagues in the database research lab Ali Kiani, Hsueh-Ieng Pai, Ahmed Alasoud, Nima Mohajerin, Deng Xi, and Celina Lian were often helpful with technical ideas and have always lent a helpful hand when I needed it. I would like to extend my special thanks to Anand Thamildurai for his diligent contributions to the FASST project and for the fruitful and joyful collaboration.

I am also grateful for the financial support provided by NSERC, Genome Quebec, and Concordia University.

Last, I would like to thank the people that I am forever indebted: my wife Maria for her patience, my son Radoslav for his contagious curiosity, and my parents for their faith in me. To them I dedicate this thesis.

Table of Contents

List of Figures

List of Tables

List of Symbols

Σ	An alphabet
$\|\Sigma\|$	The size of the alphabet (in number of symbols it contains)
$	Sequence termination symbol
b	The number of atomic elements in one disk page
B	The size of one disk page, in KB
$depth(z)$	The length of the path (the number of symbols) from ST root to node z
$\|e\|$	The edge length (in number of symbols) between parent and child node
k	The maximum number of errors allowed when matching P and S
lcp	Longest common prefix
$lp(z)$	The left pointer of a node z, $lp(z) = \min l(z) + depth(z)$
$l(z)$	The leaf set of suffix tree node z
m, $\|P\|$	The size of P (in number of symbols)
M	A simple motif
Ma	The anchor motif selected for performing structured motif search
n, $\|S\|$	The size of S (in number of symbols)
occ	The number of occurrences of P in S
p	The length of the prefixes for which the LT index is constructed
P	A pattern to be searched for
$prefixlen$	The length of the prefixes used for constructing the STTD64 index
q-x-y	A query set containing y queries, each of length x symbols
QS	A query set (containing multiple patterns P)
S	A sequence to be searched in
$S[x, y]$	The substring of S starting at position x and ending at position y, inclusive
SM	A structured motif
$SP(M)$	The Selectivity Power of simple motif M
u	Any branch node in a suffix tree
v	Any leaf node in a suffix tree
z	Any node in a suffix tree, except the root

List of Abbreviations

BLAST	Basic Local Alignment Search Tool
DNA	Deoxyribonucleic acid
EMOS	Exact Match Overlapped Structured motif search algorithm (HST-based)
EMS	Exact Match Search
EST	Expressed Sequence Tag
FASST	Fast And Scalable Search Tool (HST-based)
KMS	K-Mismatch Search
LT	Lookup Table index
LTR	Long Terminal Repeat
mRNA	Messenger RNA
NCBI	National Center for Biotechnology Information
NIH	National Institutes of Health
RNA	Ribonucleic acid
SA	Suffix Array
SMS	Structured Motif Search
ST	Suffix Tree
STEM	Suffix Tree Exact Match search algorithm (HST-based)
STKM	Suffix Tree K-Mismatch search algorithm (HST-based)
STS	Sequence Tagged Sites
STTD64	Suffix Tree, Top-Down, 64 bits index
tRNA	Transfer RNA

1 Introduction

In the past two decades, the molecular biology and genomics fields have witnessed major advances. As a result, biological data being gathered from various organisms became of enormous size and shows exponential growth rate. Bioinformatics is a relatively new research area, emerging as an interface between biological and computational sciences. Its ultimate goal is to provide the computational means needed for discovering important biological information hidden in the data, thus allowing for a better understanding of the fundamental biology of the organisms. The new knowledge obtained would have significant impacts on our everyday life, including human health, agriculture, energy sources and environment.

According to the Bioinformatics Definition Committee (National Institute of Mental Health), bioinformatics is defined as "Research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioral or health data, including those to acquire, store, organize, archive, analyze, or visualize such data" [NIH, 2000]. The National Center for Biotechnology Information (NCBI) offers a more elaborate definition of bioinformatics as "the field of science in which biology, computer science, and information technology merge into a single discipline." There are three important sub-disciplines within bioinformatics: "the development of new algorithms and statistics to discover relationships among members of large data sets; the analysis and interpretation of various types of data including nucleotide and amino acid sequences, protein domains, and protein structures; and the

development and implementation of tools that enable efficient management and processing of different types of information" [NCBI, 2007a].

The focus of this thesis is on providing efficient, scalable, and versatile processing and searching in nucleotide and amino acid sequences. It can be viewed as a "classical" bioinformatics topic, according to the definition suggested by Fredj Tekaia from Institute Pasteur: "The mathematical, statistical, and computing methods that aim to solve biological problems using DNA and amino acid sequences and related information."

Numerous bioinformatics applications use DNA and amino acid sequence data to obtain some insights and new knowledge. An example is: given a new sequence, find all sequences stored in a database which are similar (above a user-specified threshold of similarity) to it. For DNA sequences, this search can reveal important information about possibility of existence of an evolutionary relationship (homology) between a query sequence and its similar DNA sequences. For protein sequences, finding similar, already known, proteins may help in approximation of the biochemical function of the query protein. Other examples include searching the new sequence for already known motifs, expressed-sequence-tags (ESTs), sequence-tagged sites (STSs), finding exact and approximate repeats and palindromes, recognizing DNA contamination, etc. Overall, searching in DNA and protein sequence data is central to computational molecular biology, and hence performing it efficiently is highly desirable.

In general, string searching is not a new topic and has been well studied in the past. However, the nature of biological sequences and the specifics of the search operations performed on it present new research challenges. Next, we briefly describe the sources of DNA and protein sequences (Section 1.1), outline the characteristics of biological

sequence data (Section 1.2) and search operations performed on it (Section 1.3), and highlight the challenges addressed by our work (Section 1.4).

1.1 Sources of Biological Sequence Data

A DNA sequence is obtained by a procedure called DNA sequencing. It determines the exact order of the nucleotides in a segment of organism's genome (see Figure 1), where each nucleotide is represented by its base: A, C, G, or T. Because the bases exist in pairs and the identity of one of the bases in the pair determines the other base of the pair, scientists often report only one of the two complementary strands. Thus, a DNA sequence can be viewed as a long string over the alphabet {A, C, G, T}. For example, a nucleotide sequence that describes the DNA segment in Figure 1 is AGTACG. The collection of all DNA data in an organism represents its genome, which is the blueprint of this organism.

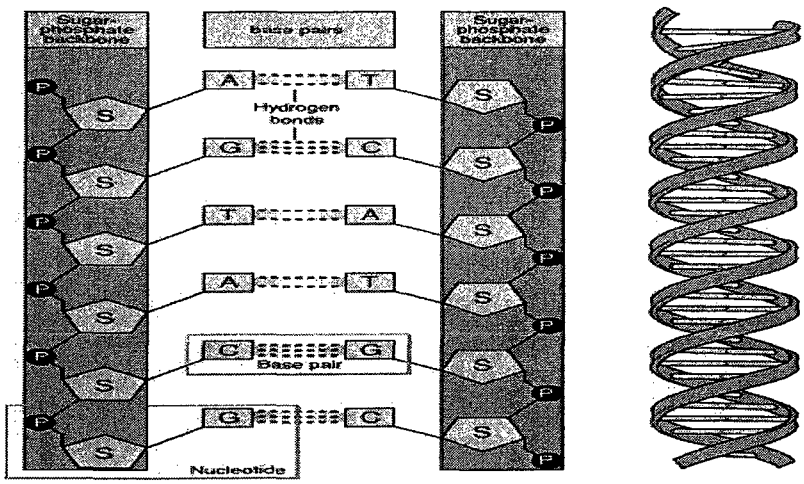


Figure 1. A DNA Segment [National Health Museum]

The other type of biological sequences that we consider in our work are protein sequences, which are synthesized from DNA in two steps, as follows. In the first step, through a process called *transcription*, a gene (nucleotide subsequence contained in

DNA), is used as a template to obtain a similar molecule, called messenger RNA (mRNA) shown in Figure 2, on the left.

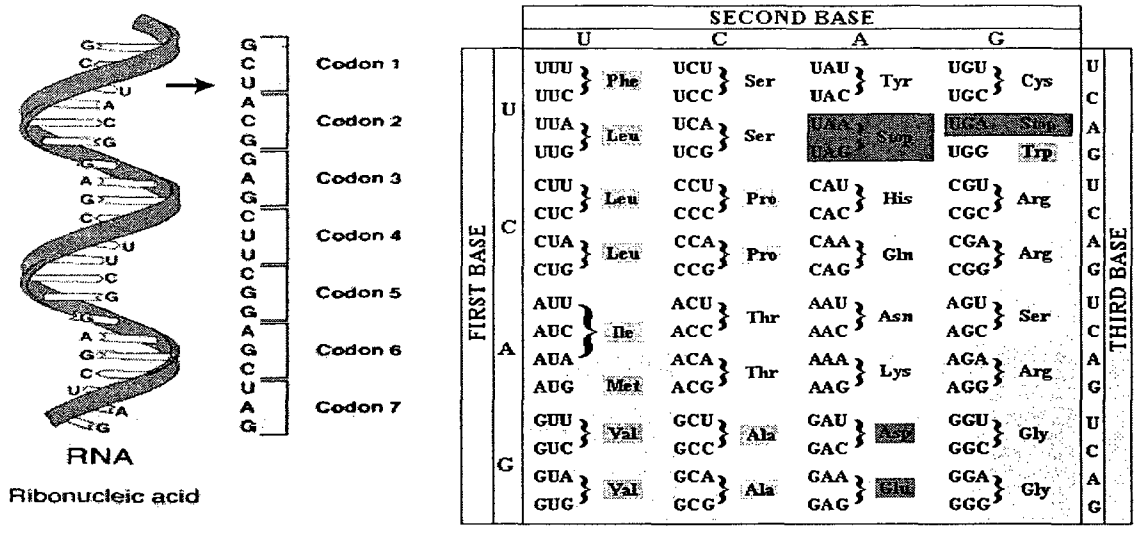


Figure 2. From DNA to Proteins [web sources]

The mRNA molecule can be represented as a string over an alphabet of size four, similar to the DNA sequence alphabet; the only difference is that T is replaced by U. The original DNA molecule stays inside the cell nucleus, but the mRNA travels out into the cytoplasm. Once there, the mRNA sequence is viewed as a list of triples, where each triple is called a codon (Figure 2, middle). In the second step in protein synthesis, called *translation*, based on the matching policy shown in Figure 2 (right), each codon is matched with one of the twenty corresponding amino acids to produce the polypeptide chain of the transfer RNA (tRNA) molecule. Peptide bonds are made between adjacent amino acids in tRNA, which eventually produces the protein sequence. For example, the mRNA string GCUACGGAGCUUCGGAGCUAG (Figure 2, left) produces the protein sequence ATGLRS*, where each amino acid is represented by its one-letter code (an alternative to the three-letter code shown in Figure 2), and the symbol * represents the stop codon. Thus, a protein sequence can be viewed as a string over the alphabet of

twenty amino acids. Due to their importance to living organisms, the proteins are often referred to as "the building blocks of life".

1.2 Characteristics of Biological Sequence Data

The first methods for DNA sequencing were developed in the mid-1970s. At that time, scientists could sequence only a few base pairs per year. By 1990, only a few laboratories had managed to sequence 100,000 bases, and the cost of sequencing remained very high. Since then, technological improvements and automation have provided increased speed and decreased the cost to the point where individual genes can be sequenced routinely, and some labs can sequence well over 100 million bases per year.

An important milestone in this aspect was achieved by the orchestrated efforts of the Human Genome Project. On April 14, 2003 the National Human Genome Research Institute, the U.S. Department of Energy, and their partners in the International Human Genome Sequencing Consortium announced the successful completion of sequencing the entire human genome, resulting in a DNA dataset which contains sequences of total length about 3 billion nucleotides.

Other enormous DNA and protein datasets were collected as well and were made publicly available. For instance, as of July 2007, the Trace Archive dataset [Trace, 2007], which records single-pass DNA sequencing reads coming from large-scale sequence projects, contains above 1.5 billion DNA sequences of total size more than 800 billion nucleotides, and its size doubles almost every 10 months. The TrEMBL protein dataset [TrEMBL, 2007], as of October 2007, contains almost 5 million protein sequences, of total size more than 1.6 billion amino acids, which also exhibits exponential growth. GenBank [GenBank, 2007a] is a comprehensive database that contains publicly available

DNA sequences which represent both individual genes and partial and complete genomes for more than 240,000 named organisms, obtained primarily through submissions from individual laboratories and batch submissions from large-scale sequencing projects. As of August 2007 [GenBank, 2007b], GenBank contains more than 70 million DNA sequences of total size around 80 billion nucleotides, and its size doubles almost every 17 months. Overall, there are already enormous collections of biological sequence data publicly available and there is a stable trend indicating that their size will continue to grow considerably in the foreseeable future.

Next, we highlight some characteristics of DNA and protein sequences, which make them distinguishable from other types of sequence data, such as natural language texts, times series, etc.

The most important characteristic of biological sequence data is that it does not have a structure other than being a string. That is, DNA and protein sequences cannot be meaningfully broken into parts/words. This explains why standard database technology which views the data as tuples with attributes is not applicable. For the same reason, the existing solutions from the natural language processing field are not effective in this context in general. New techniques are required for fast, scalable, and versatile processing of biological sequences.

Another feature of biological sequence data, especially evident for DNA sequences, is the wide range of the sequence sizes. While the size of the protein sequences is usually up to several thousand amino acids, the range of DNA sequence sizes is between several hundred nucleotides (e.g., ESTs) up to several billions (e.g., the entire human genome). Improving the efficiency of existing techniques for searching in shorter sequences would

have a practical impact, considering the query volume in this domain. However, a real challenge is development of suitable techniques to support efficient search in long sequences, in addition to the one implied for short sequences.

Also, it is desirable to provide a uniform technique for handling both DNA and protein sequences, since from biological point of view these two types of data are closely related to each other. One difference between DNA and protein sequences which has to be accounted for is their different alphabet size – 4 for DNA and 20 for protein sequences. The performance of some of existing string searching techniques is highly dependable on the alphabet size, thus rendering them inappropriate as a uniform solution.

1.3 Characteristics of Search Operations

Upon obtaining a new DNA or protein sequence, it is natural that a biologist would like to learn as much as possible about it and there are various search tasks that could be performed. For instance, one direction is to perform sequence similarity search in existing databases, such as GenBank and TrEMBL. As explained, the results of this type of search can be used to infer functional and evolutionary relationships between sequences, and help identify members of gene families. Another direction for investigating the properties of a new sequence is to search in it for some previously known, relatively short sequences whose biological function is already established. One such example is searching for ESTs, whose presence or absence helps biologists in gene discovery and sequence determination tasks.

A core component in numerous search applications for biological sequences is to find substrings in the sequence data which are same (exact match) or similar (approximate match) to a query pattern. Next, we briefly describe three such search operations.

The simplest case is *exact match search* - given a query pattern P and a sequence S, the goal of exact match search is to find all substrings in S which match exactly P and report their starting positions.

In some cases, an *approximate match* is acceptable and sometimes even desirable, accounting for evolutionary changes. The goal of an approximate search is to find all substrings in S which match P with at most k errors and report their starting positions. The notion of matching errors between two strings S_1 and S_2 is defined based on the notion of edit operations [Levenshtein, 1966]. Any string S_i can be transformed into another string Sj based on three edit operations: insert, delete, and replace, on individual characters of S_i. The edit distance between S_i and S_j is defined as the minimum number of edit operations needed to transform S_i into S_j. For example, consider the DNA strings S_1 = ACTTAGC and S_2 = AATGATAG. We note that S_1 can be transformed into S_2 performing four edits: one replace (R), two inserts (I), and one delete (D), as shown in Figure 3.

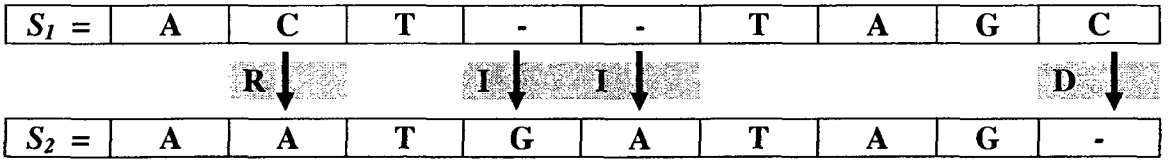


Figure 3. Edit Operations and Approximate Matching

A *k-match* between two strings S_i and S_j is observed if S_i can be transformed to S_j with at most k edit operations. Exact match search is a special case of k-match problem, where $k = 0$. For $k > 0$, we distinguish two types of approximate search - the *k-difference search* and its simpler variant, the *k-mismatch search*. The former allows using all three edit operations, while the latter allows only the replace operation. In other words, k-mismatch

requires that the size of P and the size of the matching substring from S to be the same, whereas k-difference allows gaps to be introduced in either P or S.

As a result of the expansion in the understanding of biological processes, novel search operations become relevant and find their way into search tools. For example, it was noted that long terminal repeat (LTR) retrotransposons have a significant presence in the typical mammalian genome and are believed to have major impact on genome structure and function [Feschotte *et al.*, 2002; McCarthy and McDonald, 2003]. Thus, the known LTR retrotransposons can be represented as structured motifs (explained later), and a new sequence could be searched for them. Another application of structured motif search is to search for composite regulatory binding sites in DNA sequences, which helps understanding the complex transcriptional regulatory network in Eukaryotic organisms.

Last, as a result of the advances in computational power and Internet connectivity, more and more biologists around the world use the available online query services on a daily basis.

In summary, investigating the properties of a new sequence involves performing various types of search tasks and they should be done efficiently, on the vast amount of sequence data available. There is a rapid growth in both versatility and volume of the search operations required in this process.

1.4 Challenges Ahead

To illustrate the motivation for our work, let us use as an example BLAST [Altschul and Gish, 1990; Altschul *et al.*, 1997], a popular bioinformatics tool for searching databases of already known sequences for those similar to a new (query) sequence. BLAST is available in two variants. The online variant, NCBI-BLAST, allows for submitting a user

query to the BLAST interface hosted at the NCBI site [NCBI, 2008]. Upon receiving the query, it is transmitted to a cluster of several hundred computers, which use their main memory exclusively for segments of a database to be searched. The search is carried out on each of these machines by performing a full database scan, the result of each machine is returned back to the interface, aggregated, and then presented to the user. An advantage of this approach is that the computational power required at the client site is ubiquitous – any computer with Internet connection and a browser will be sufficient. On the other hand, the amount of available data, the number of search operations performed, and the BLAST algorithmic framework put considerable demands for resources at the server site. As a result, the NCBI-BLAST search time for a single query almost doubles each year [Chen, 2004]. Considering the fact that NCBI-BLAST receives daily hundreds of thousands search requests (the exact number varies between different sources from 142,822 [Cameron, 2006] to 400,000 requests [Ostell, 2005]), this decrease in the performance has an undesirable practical impact. Another issue here from the user's point of view is privacy/ownership of the data sent as a query to the online BLAST interface.

To address some of these shortcomings, BLAST is also offered as a standalone program, which can be freely downloaded and installed on a local computer. Although this approach solves the privacy/ownership concern, it results in poor performance caused by the decreased computational power a typical user has. In [Cameron, 2006], authors use the standalone version of BLAST on a regular desktop computer (Intel Pentium 4, 2.8GHz, 2 GB RAM). One of their goals was to estimate the time needed for performing 142,822 searches in the corresponding databases. While this search task took 24 hours at NCBI-BLAST, the estimated time for the standalone version was at least 93 days!

Another study [Hunt *et al.*, 2002] of the performance of standalone BLAST was conducted on a more powerful SUN Enterprise 450 machine with 4 processors, 2 GB RAM, running Solaris 7 OS. A database consisting of 3 human chromosomes (294 Mbp, 10% of human genome, in main memory) was searched for 99 query sequences (predicted human genes) with lengths between 429 to 5,999 nucleotides. A total of 6,559 similar sequences were found. The time for completing the search was 62 hours!

These examples illustrate two important challenges. First, it cannot be expected that the problem of providing efficient search will be solved by pure hardware means in the foreseeable future. Figure 4 provides a comparison between the growth rate of RAM memory available on typical computers and the growth rate of GenBank, a biological sequence database. As can be seen from the figure, there is an increasing disparity between these two rates. Accepting the fact that searching in biological sequences would require utilization of secondary memory raises the challenge of developing solutions that are disk I/O efficient.

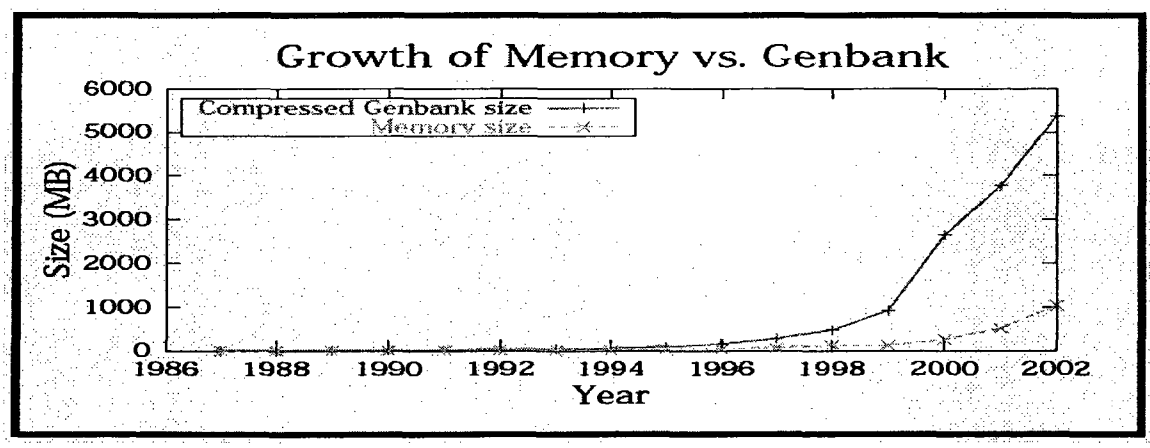


Figure 4. Growth of Computer Main Memory vs. GenBank Size [Darling and Feng, 2002]

Second, even if the RAM size is large enough to fit the data sequence in the memory, existing tools (including BLAST) which use full data scan lack computational efficiency and are inadequate for processing the large variety and volume of search operations

required in current bioinformatics applications.

It is important that these two challenges are properly addressed. Many of the search tasks are conducted as a preliminary step in analyzing sequence data, and the search results are used to guide the subsequent wet-lab experiments. Naturally, the search techniques should be efficient, so that the cost of initially investigating and testing more hypotheses remains reasonable.

We investigate, both theoretically and practically, appropriate models for processing and searching biological sequence data. Traditional database management systems although successful for relational data (i.e., tuples with attributes) are not suitable for handling sequence data. We aim at developing an appropriate management system specifically for biological sequences to support efficient, scalable, and versatile search operations in biological sequences. Also, we believe that a solution that requires off-the-shelf computational resources is desirable, since it will be accessible to a wider range of potential users.

The developments and contributions in this thesis provide steps towards the realization of the idea. First, we propose an index representation, called HST, suitable for biological sequences (Section 3.5) together with its construction algorithm (Section 4.3). Second, we implement a HST based solution for exact (Section 5.1) and approximate k-mismatch search (Section 5.2), which are at the core of numerous bioinformatics applications. We also implement a HST based solution for structured motif search, a practical bioinformatics application (Section 5.3). Third, by conducting extensive experimental evaluation, we investigate and compare the performance of the HST based solutions to existing state-of-the-art techniques. Our experimental results (Chapter 6)

indicate that the proposed solution exhibits several important characteristics – efficiency, scalability, versatility, and affordability, running on a typical desktop computer.

The rest of this thesis is organized as follows. In Chapter 2, we provide the necessary background, review related work, explain our choice of using suffix tree as a promising data structure for our index, and formally state the problems addressed by this work. In Chapter 3, we present the HST index and discuss its advantages and disadvantages over other suffix tree based representations. The HST construction algorithm is discussed in Chapter 4. In Chapter 5, we present three search algorithms which use HST to provide solutions to the exact match, k-mismatch, and structured motif search problems. The results of our experiments for evaluating and comparing the performance of our technique to the best-known alternatives are presented in Chapter 6. Conclusions and an outline of the future work are provided in Chapter 7.

2 Background and Related Work

In this chapter, we review work related to exact and approximate search in sequence data. We describe three approaches and their characteristics. We then discuss why we chose to base our solution on suffix trees as a promising data structure for indexing biological sequences.

Let P be a query sequence consisting of m symbols, i.e., $|P| = m$. Let S be a data sequence to be searched, where $|S| = n$. Note that m is usually much smaller than n. From this point on, throughout the thesis we refer to P as *query* and to S as *data*. The result of the search is the set of start positions of substrings in S that match P with at most k errors; $k = 0$ for exact match search. We can classify the existing search techniques into three general approaches: no preprocessing, preprocessing of P, and preprocessing of S. In the following subsections, we describe these approaches and illustrate each approach by example.

2.1 No Preprocessing

To perform exact match search, the no preprocessing approach, also called the *naive method*, aligns the left end of P with the left end of S and compares the sequences P and S character by character, going from left to right until a mismatch is found or P is exhausted. If no mismatch is detected, an occurrence of P in S is found and reported. Next, no matter what the result of this comparison is, P is shifted one position to the right, and the comparison process is repeated. The search completes when the right end of P passes the right end of S. By allowing up to k mismatches to occur in the comparison process, this method is easily extendable to handle k-mismatch search. This is the

simplest method to understand and implement, but its running time is $O(mn)$ and it does not scale up for long data and query sequences, which we deal with in biological data domain.

The k-difference search can be implemented based on dynamic programming (DP). A direct application of DP runs in $O(mn)$ time and requires $O(mn)$ space. Myers [Myers, 1986] improved the time and space complexity to $O(kn)$, where k is the number of errors, usually much smaller than m, by maintaining only the required part of the distance matrix. For small values of k, this technique is expected to be reasonably efficient. However, regardless of the theoretical time bound, its actual implementations are too slow [Navarro, 2001]. In summary, for long data sequences and higher values of k, this approach is inefficient or even infeasible in terms of time and space.

2.2 Preprocessing the Query

For exact match search, popular algorithms which preprocess the query P include Knuth-Morris-Pratt algorithm [Knuth *et al.*, 1977], the Boyer-Moore algorithm [Boyer and Moore, 1977], and its Apostolico-Giancarlo version [Apostolico and Giancarlo, 1986]. The main idea here is to preprocess P and gather some information which will allow shifting P more than one position to the right, thus reducing the total number of character comparisons performed. These algorithms are comparison based and run in $O(m + n)$ time and $O(m)$ additional preprocessing space.

Considering approximate search, algorithms that preprocess P to build finite automata (FA) generated significant research interest [Ukkonen, 1985; Wu *et al.*, 1996; Navarro, 1997]. In [Baeza-Yates and Navarro, 1999], the authors proposed a non-deterministic finite automaton (NFA) based solution to the problem as follows. Using the query

sequence P, an NFA with $(k+1)(m+1)$ states is constructed in time $O(km)$. The NFA sequentially reads S and reaches acceptance state when a substring is processed with at most k errors. The search time is $O(n)$ for short queries (i.e., $km = O(\log n)$). Longer queries are handled differently depending on the value k. For smaller k values, the query is partitioned into computer words of length $w = \Omega(\log n)$, and the search is performed in $O\left(n\sqrt{km/w}\right)$ time. For larger k values, the automaton is partitioned, in which case the search is done in $O(n\lceil km/w \rceil)$. The experimental results show that this technique performs well for short queries and small alphabets. However, its performance deteriorates for longer queries, as well as for data sequences S which do not fit in the main memory.

In summary, the query preprocessing approach achieves considerable search time improvement compared to the no preprocessing approach. The main disadvantage of automata-based solutions is that the required space is exponential on either m or k, which limits their applicability [Navarro, 2001]. Another problem of a FA-based solution is that it depends on a sequential scan of S. As a result, the search time is proportional to the size of S, which could be very large. Moreover, for large sequences which do not fit in memory, the sequential scan of S may lead to redundant I/O operations, as parts of the sequence which are irrelevant to a particular query will also be read from disk anyway. Also for this approach, the preprocessing time is incurred by the user when he/she poses a query, noting that the result of preprocessing P can not be utilized for subsequent queries.

2.3 Preprocessing the Data

A basic idea to speed up the search tasks in sequence data is to create and use a suitable index. A clear advantage here would be that the index construction is independent of user queries and is done offline. As a result, the cost of index construction is not incurred by the user and is amortized over many queries, since biological sequences are relatively stable and updates are rare.

This approach has proved its potential for some types of sequence data, especially natural language texts. The cost of creating an index is quickly amortized by performing much faster searches. However, developing suitable index structures for biological sequences to support various types of search tasks efficiently is an active research topic [Navarro, 2001]. There are some important challenges that should be addressed.

First, as mentioned in Section 1.2, biological sequences cannot be broken down into words/parts, as in natural language texts. This explains why some known indexing techniques such as inverted files [Witten *et al.*, 1999] and prefix index [Jagadish *et al.*, 2000] successful in other domains are not suitable for biological sequences.

Second, some biological sequences can be of considerable length. For example, chromosome 2, the longest human chromosome, contains approximately 240 million nucleotides. If one is to consider the entire human genome as one single string, obtained by concatenation of all 24 human chromosomes (later we will explain why), the result is a sequence of size above 2.8 billion nucleotides. Similarly, considering the whole TrEMBL protein collection [TrEMBL, 2007] as a single string results in a sequence of size 1.3 billion amino acids. These numbers raise issues related to index size, index construction time, and even the ability of index construction algorithms to handle

sequences of such sizes.

Third, due to the increasing number of emerging search applications, there is a growing need for more advanced indexing techniques to support versatile search operations in biological sequence data.

The String B-tree index [Ferragina and Grossi, 1999] addresses two important challenges. First, its exact match search time complexity is $O(m + \log n + occ)$, where occ is the number of exact occurrences of P in S, a significant improvement compared to the $O(m + n)$ time bound for the query preprocessing approach. Second, the exact match search algorithm using this index performs $O((m + occ)/b + \log_b n)$ disk accesses in the worst case, where b is the number of elements in one disk page, which was proved to be asymptotically optimal for large alphabets whose characters can only be accessed by comparisons. However, String B-Tree index was not adopted in bioinformatics search tools, since it was designed for exact match search only and hence does not provide efficient support to approximate search tasks [Hunt *et al.*, 2002; Yang *et al.*, 2004]. Further, the exact match search time complexity provided by String B-Tree can be further improved, as discussed later.

The q-grams index [Navarro and Baeza-Yates, 1998] was originally proposed for supporting approximate search in natural language texts. The q-grams technique is based on a similar idea to the one in BLAST; the difference is that while BLAST preprocesses P, the q-grams index is built on S. The complexity of k-mismatch search using q-grams index is $O(m^2(k + \log n) + cm^2)$, where c is the number of candidate answers which have to be verified. The exact match search is seen as a special case of k-mismatch search and the q-grams index provides efficient solution to it. Several works adopted the q-grams

index for searching in biological sequence data, e.g., [Burkhardt *et al.* 1999; Giladi *et al.*, 2000; Navarro *et al.*, 2000]. However, there are some disadvantages of q-grams index in this context. First, for higher k values and longer queries, the number of verifications performed for the intermediate results increases. The verification step is expensive, since it requires random access to S which translates to a significant number of disk I/O operations, rendering q-grams index not suitable for low similarity matches [Navarro, 2001]. Second, the search time bound given above relies on using optimized partitions of P, whose generation requires additional time and space. Finally, the applicability and efficiency of q-grams index to k-difference search task is not clear and to the best of our knowledge is still an open research question.

In the past decade, there has been an increasing interest in suffix trees (ST) and suffix arrays (SA) as suitable data structures for indexing biological sequences. Next, we discuss advantages and drawbacks of these two approaches for processing biological sequence data.

2.3.1 Suffix Trees

Suffix Trees (ST) are powerful index structures for sequence data, which recently have attracted considerable research attention, including [Gusfield, 1997; Delcher *et al.*, 1999; Burkhardt *et al.*, 1999; Kurtz and Schleiermacher, 1999; Hunt *et al.*, 2002; Giegerich *et al.*, 2003; Policriti *et al.*, 2004; Bedathur and Haritsa, 2004, Cheung *et al.*, 2005; Tian *et al.*, 2005; Phoophakdee and Zaki, 2007].

One advantage of ST index is its efficiency in supporting various search tasks, including exact match, k-mismatch, and k-difference search. For example, exact match search can be done in $O(m + occ)$ time [Gusfield, 1997]. It is remarkable that the search

time depends only on the query size and the number of its occurrences, and is independent of the length of S. Both the k-mismatch and k-difference search tasks can be done in $O(kn)$ time using a ST index [Gusfield, 1997]. Although the time complexity of k-difference search utilizing a ST index is the same as for the no preprocessing approach (Section 2.1), which is based solely on dynamic programming (DP), it was noted that hybrid algorithms which use a ST index to solve some of the related subproblems exhibit superior practical performance than those based solely on DP [Gusfield, 1997].

Another advantage of the ST index is its versatility. While most of the indexing techniques discussed so far are developed with particular exact or approximate search operation in mind, ST can be used to support other search tasks - Apostolico [Apostolico, 1985] gives more than 40 references; Gusfield [Gusfield, 1997] discusses more than 20 applications of ST in the area of bioinformatics. Examples include searching for various types of repeats in a single sequence, finding the longest common substring/subsequence of several sequences, shortest superstring problem, etc. As a result, ST have found practical applications in a number of bioinformatics tools, including MUMmer [Delcher *et al.*, 1999], QUASAR [Burkhardt *et al.* 1999], REPuter [Kurtz and Schleiermacher, 1999], and SMaRTFinder [Policriti *et al.*, 2004]. MUMmer is a system for aligning genome size sequences. It builds a generalized ST (a ST for more than one sequence) to find maximal unique matches. QUASAR is a homology search tool, which given a set of sequences builds a ST for one of them and counts the number of exact matching seeds from the other sequences. If the number of seeds for a region is more than a predefined threshold, this region is searched using BLAST. The REPuter program family provides software solutions to compute and visualize repeats in chromosome and genome

sequences. It builds a ST for the sequence and finds repetitions on-the-fly. SMaRTFinder builds a ST index for a data sequence *S*, and uses it to search for the presence or absence of previously known structured motifs in *S*.

However, there are two major disadvantages of ST index. First, even though the size of the ST index is linear to the size of the data, it is considerably large. To the best of our knowledge, the most space efficient (non-compressed) implementation of a ST requires 8.5 bytes on average per each character in *S*, and 12 bytes per character in the worst case [Giegerich *et al.*, 2003]. While secondary storage is relatively cheap and abundant, the large index size introduces the second, more important shortcoming of suffix trees, discussed next.

Throughout this thesis, we refer to construction and search algorithms that require both the sequence and its whole index to fit simultaneously in main memory as *memory-based*, and to algorithms that relax this requirement as *disk-based*. Even for the most space efficient ST representation [Giegerich *et al.*, 2003], the ST index sizes necessitate either large main memory or efficient disk I/O construction and search algorithms. For example, it is estimated that a memory-based construction of the ST index for the entire human genome of size approximately 3 GB would require about 45 GB of RAM [Kurtz, 1999]. Historically, the first suffix tree construction algorithms were memory-based [Weiner, 1973; McCreight, 1976; Ukkonen, 1995]. They build the index in time linear to the size of data *S*, but are limited by the size of available main memory. As a result, until recently, the second disadvantage of suffix trees was the upper bound on the size of the sequence ST construction algorithms can handle. Recent research, including Hunt [Hunt *et al.*, 2002], TOP-Q [Bedathur and Haritsa, 2004], DynaCluster [Cheung *et al.*, 2005],

TDD [Tian *et al.*, 2005], and Trellis [Phoophakdee and Zaki, 2007] focus on disk-based ST construction algorithms, which by efficiently utilizing the secondary memory provide the opportunity to build the ST index for long sequences (e.g., 3GB) with reasonable amount of RAM memory (e.g., 2GB), at the cost of increased time complexity (e.g., $O(n^2)$). However, these proposed indexing techniques often focus on index construction and storage cost, while the efficiency and scalability of the search based on these indexes is given less research attention.

Addressing the issues related to the search performance of a ST based index is at the core of this thesis. We propose a ST based index that has reasonable storage requirements, is suitable for disk-based construction, and most importantly, provides efficient, scalable, and versatile support to search applications. Our proposed HST index is presented in Chapter 3, its construction algorithm in Chapter 4, and search algorithms based on it in Chapter 5.

2.3.2 Suffix Arrays

The problem of indexing sequences within the available main memory was addressed by Manber and Myers [Manber and Myers, 1993] in a different way, by introducing a space efficient index structure, called suffix array (SA). A *basic* SA [Manber and Myers, 1993] is essentially a list of pointers to the lexicographically sorted suffixes in sequence S, requiring only $4n$ bytes for sequences of up to 2^{32} symbols. The time complexity of basic SA index construction is $O(n \log n)$. The smaller SA index allows for memory-based SA index construction for sequences that would otherwise require a disk-based, thus slower, ST index construction, which is a major advantage of SA. For longer sequences, there are

also several disk-based SA construction algorithms [Crauser and Ferragina, 2002], most notably DC3 [Dementiev *et al.*, 2005] and LOF-SA [Sinha *et al.*, 2008].

However, there are two major disadvantages of the basic SA index. First, the exact match search using the basic SA has $O(m \log n + occ)$ time complexity. One way to improve this search performance is by constructing and using an additional table, called *longest common prefix* (*lcp*) table [Manber and Myers, 1993], which is of size n. Using the *lcp* table, the exact match search is conducted in $O(m + \log n + occ)$. Recall that the same problem can be solved in $O(m + occ)$ using ST. The difference in the search complexity between SA with *lcp* table and ST based solutions becomes even wider for more involved search tasks such as approximate search.

Second, basic SA exhibit poor locality of reference when used by disk-based search algorithms, due to the binary search of the suffix array [Sinha *et al.*, 2008]. For longer sequences, this poor locality of reference leads to inefficient disk I/O operations and to unsatisfactory search performance.

To address the search complexity disadvantage, [Abouelhoda *et al.*, 2004] proposed *enhanced* suffix array (ESA), which is a collection of tables, including the basic SA and an *lcp* table similar to [Manber and Myers, 1993]. ESA provides versatile and efficient support for more than 10 search applications, with the same time complexity as ST based solutions [Abouelhoda *et al.*, 2004]. However, its space requirement increases to $12n$ for storing the additional tables. It should be noted that not all these tables are needed in main memory during index construction or search operations. As a result, although the total size of ESA index is comparable to a ST based index, in general the memory-based ESA technique can handle longer sequences compared to memory-based ST solutions.

Vmatch [Vmatch, 2007] is a commercial software tool (free for academic use) that implements ESA index. It has a theoretical limit of 400MB for sequences it can handle on 32-bit desktop computers [Vmatch, 2007] and being a memory-based technique, a practical limit of about 250MB on our desktop with 2GB RAM. The experimental results for exact match search reported in [Abouelhoda *et al.*, 2004], show that Vmatch outperforms the basic SA with *lcp* table solution in [Manber and Myers, 1993] and the ST based solution in [Kurtz, 1999]. For longer sequences, a 64-bit version of Vmatch that runs on large server class machines is proposed, but it is not included in our comparison study, as we focus on typical desktops. Due to its efficient construction and search algorithms, and for its capability to handle versatile search tasks, we consider Vmatch as the best known solution for sequences up to 250 MB it can handle.

SeqAn library [SeqAn, 2008] provides another implementation of ESA indexing technique [Abouelhoda *et al.*, 2004]. It offers memory-based index construction and search for short sequences, as well as disk-based algorithms for longer sequences, which allow handling sequences up to 4GB on typical desktops.

For such long sequences, DC3 is shown to be superior to earlier disk-based SA construction techniques [Dementiev *et al.*, 2005], but it is designed for multiprocessor machines and does not perform as well on a single processor computer. On such a computer, it has been shown that DC3 is inferior to TDD in construction time [Tian *et al.*, 2005]. We do not consider DC3 in our comparison since it generates only the basic SA, without the additional tables used (e.g., in ESA) to improve search performance.

Several works address the basic SA locality of reference disadvantage, an important issue for disk-based solutions. [Baeza-Yates *et al.*, 1996] propose a two-level index,

which consists of disk-resident basic SA and memory-resident index, called SPAT, derived from and containing some information about the SA. In the first phase of the search, the algorithm uses only the SPAT index to reduce the number of disk I/O operations performed for accessing the disk-resident SA in the second phase. The experimental results show a significant search time improvement (about 10 times on average) compared to basic SA.

A recent work [Sinha *et al.*, 2008] adopted the idea of a two-level index arrangement and proposed LOF-SA, which consists of small LOF-trie (a heavily truncated version of suffix trie) and large LOF-array (an interleaving of the basic SA and the *lcp* table) of total size $13n$. The search starts by traversing the LOF-trie in a top-down manner. If all query characters are matched during this traversal, the set of query occurrences is obtained by a simple scan of the region of the LOF-array determined by the pointers of the last verified trie node. Otherwise, the search continues for the remaining query characters using both the LOF-array and the sequence. The experimental results in [Sinha *et al.*, 2008] for exact match search in sequences up to 2 GB show superiority of LOF-SA over SPAT [Baeza-Yates *et al.*, 1996] and the ST-based TDD [Tian *et al.*, 2005] and Trellis [Phoophakdee and Zaki, 2007].

2.4 Summary of Related Work

Our review of various techniques for processing biological sequence data indicates the clear advantage of the approach which preprocesses the data sequence and builds an index. Due to the nature of biological sequences, the indexing techniques proposed for general sequence data, such as inverted files [Witten *et al.*, 1999] and prefix index [Jagadish *et al.*, 2000] are not suitable. Further, since it is important for the index to

provide efficient support for various search applications, we refrain from adopting indexing techniques, which provide support for only some of the search tasks, such as q-grams [Navarro and Baeza-Yates, 1998] and String B-Tree [Ferragina and Grossi, 1999].

Recently, significant advances have been made in developing techniques suitable for handling long, genome size sequences, e.g., [Crauser and Ferragina, 2002], [Bedathur and Haritsa, 2004], [Cheung *et al.*, 2005], [Tian *et al.*, 2005], [Dementiev *et al.*, 2005], [Phoophakdee and Zaki, 2007], [Sinha *et al.*, 2008]. However, the proposed indexing techniques often focus on index construction and storage cost, while the efficiency and scalability of the search based on these indexes is given less research attention. In order to provide informative comparison of these techniques, a more rigorous evaluation of their search performance is necessary.

We choose to base our solution on suffix tree (ST) index, due to the efficient and versatile support it provides to a number of bioinformatics search tasks, as shown theoretically in [Gusfield, 1997]. However, in order to provide practical solutions based on a ST index, the following three important aspects must be considered: ST representation and storage space (Chapter 3); a disk-based construction algorithm (Chapter 4); and the I/O efficiency of the disk-based search applications based on the index (Chapter 5).

In this work, we study the construction cost of our proposed HST index, as well as the exact match, k-mismatch, and structured motif search performance of the algorithms based on HST, and compare them to the best known existing solutions which support the particular task.

We compare the index construction cost of HST to the one for the suffix tree based TDD and Trellis, as well as to the ESA based Vmatch and SeqAn.

For exact match search we consider HST, Trellis, TDD, Vmatch, and SeqAn, since all of them support this search task. Unfortunately, the source code of LOF-SA is not available yet [Sinha, personal communication], so we could not make a direct comparison in this work.

For k-mismatch search we compare the performance of HST and Vmatch, the only two among the considered techniques that support such functionality at this time. TDD has a k-mismatch search algorithm; however it produces run time errors.

Finally, we compare the performance of our HST based structured motif search with SMOTIF1 [Zhang and Zaki, 2006], the best known existing solution (discussed in Section 5.3) since it outperforms the suffix tree based SMaRTFinder [Policriti *et al.*, 2004] and at this stage the other considered techniques do not support such functionality.

2.5 Problem Statement

As a result of our investigation of various indexing techniques for biological sequence data we chose to study in more detail the suitability of suffix trees (ST) as a powerful index to support major core search operations. Our goal is to develop a ST index that supports efficient, scalable, and versatile search on a typical desktop computer. The challenges of this research can be grouped into three major categories:

(1) ST Representation – deciding the data model and the information it stores (Chapter 3);

(2) ST Construction – developing a disk-based ST index construction algorithm, which is efficient in terms of construction time and scales up with respect to the size of the sequence being indexed (Chapter 4);

(3) Search Applications – developing a set of core search algorithms which use our ST index and perform efficient disk I/O operations (Chapter 5).

Note that our goal is to develop solutions for typical desktop computers, such as ours: 32-bit architecture, single processor Intel Pentium 4 @ 3GHz, 2 GB RAM, 300 GB HDD, 2 MB L2 cache, running Linux kernel 2.6. This will make our proposed techniques accessible and used by a wider range of users in bioinformatics community.

3 Suffix Tree Representations

A ST for an n character sequence S is a rooted directed tree with n leaf nodes numbered 1 to n. Each branch node has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have labels which start with the same character. As a result, the number of branch nodes in a ST varies from $n/|\Sigma|$ to n, where $|\Sigma|$ is the alphabet size. The key feature of a ST is that for any leaf node v, the concatenation of the edge labels on the path from the root to node v spells out the suffix of S that starts at position v, i.e., $S[v..n]$.

In order to illustrate the various ST representations, throughout this chapter we consider the following sample sequence S = ATGATATGTGAAATAGTAGA\$. The symbol \$ does not belong to the sequence alphabet and is used as a termination symbol, ensuring that no sequence suffix is a prefix of any other suffix. In Figure 5 we show a high-level, graphical representation of a ST for S. The numbers in squares are used to enumerate the ST nodes in the top-down, left-to-right order in which they are evaluated and recorded. Each edge is labeled with the corresponding characters from S. The number below each leaf node v illustrates the starting location in S at which we can find the suffix indicated by the edge labels on the path from the root to v.

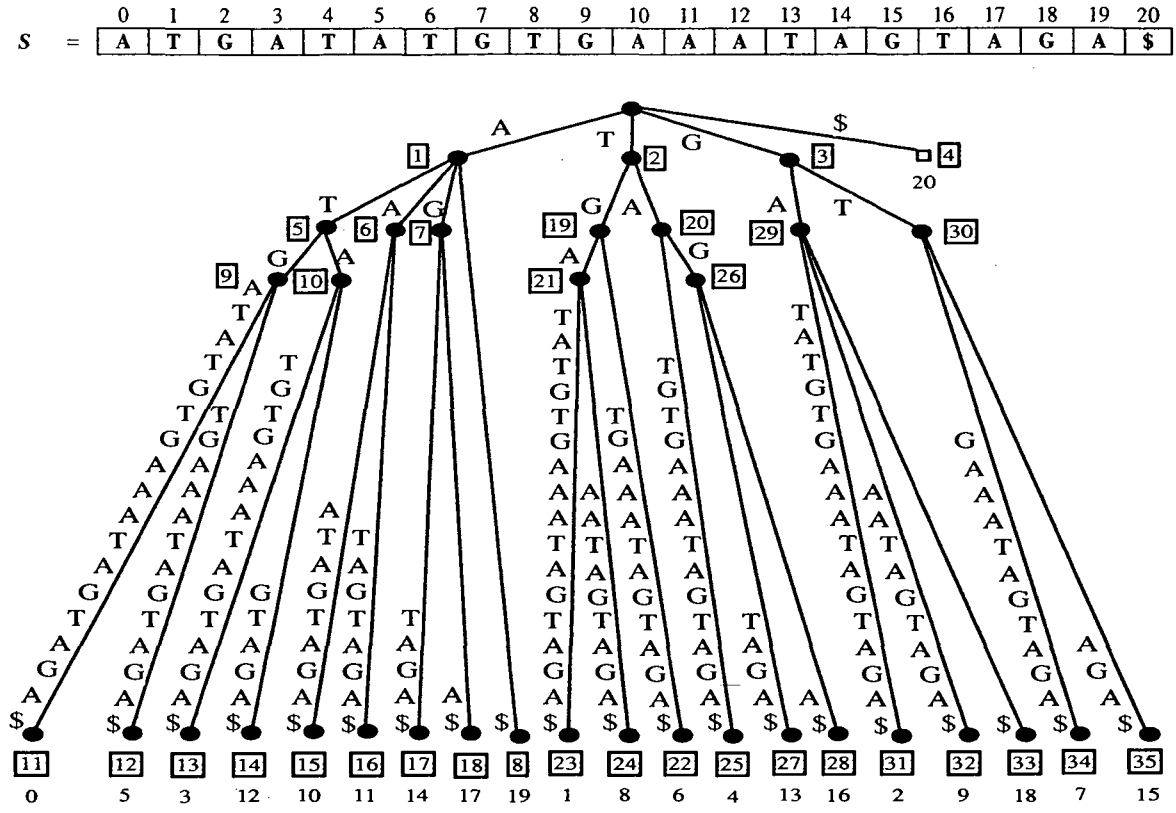


Figure 5. Graphical Representation of a ST for S = ATGATATGTGAAATAGTAGA$

Important design issues related to an actual ST representation include what information to store as well as the layout of the nodes. There are different ST representations introduced, including LC-trie [Andersson and Nilsson, 1995] , hash table [Kurtz, 1999], linked list [Kurtz, 1999; Dorohonceanu and Nevill-Manning, 2000; Hunt *et al.*, 2002], truncated suffix tree [Ristov, 2003], suffix sequoia [Hunt, 2003], suffix binary search tree and suffix AVL Tree [Irving and Love, 2003], *wotd* [Giegerich *et al.*, 2003], TDD [Tian *et al.*, 2005], and Trellis [Phoophakdee and Zaki, 2007]. In this chapter, we first review some of the most relevant ST representations and highlight their advantages and shortcomings. We will then present our proposed ST representation, which we called HST.

3.1 *wotd* ST Representation

The *wotd* ST representation [Giegerich *et al.*, 2003] derives its name from the corresponding ST construction approach called Write Only Top Down. For a leaf node v in a ST for a sequence S, the *leaf set* of v, denoted $l(v)$, contains the position v in sequence S at which we can find the suffix denoted by the edge labels from the root of ST to v. For example, for leaf node 13 in Figure 5, we have $l(13)=\{3\}$. For a branch node u in a ST, the leaf set of u is defined based on the leaf sets of the children of u, i.e., $l(u) =\{l(v)|$ *v is a leaf node in the subtree rooted at u*$\}$. For instance, for node 10 in Figure 5, its leaf set would be $l(10)=\{l(13), l(14)\}=\{3, 12\}$. There is a total order, $\prec$, defined on the nodes in the tree, as follows. For any pair of nodes x and y, which are children of the same node u, $x \prec y$ if *min* $l(x) <$ *min* $l(y)$. In Figure 5 for example, we have that node 1 $\prec$ node 2, since *min* $l(1) = 0$ is less than *min* $l(2) =1$. The *depth* of a node z, *depth* (z), is defined as the length of the path (in number of characters on the edge-labels) from the root to the parent of z. For example, *depth* (26) = |TA|, i.e., *depth* (26) = 2. For any ST node z, its *left pointer,* denoted by $lp(z)$, is defined as *min* $l(z)$ + *depth* (z). For example, $lp(6)$ = *min* $l(6)$ + *depth* (6) = 10 + 1 = 11.

Note that in the *wotd* representation, the children of a branch node are ordered based on the position in S of the strings encoded from root to each child. For example, node 19 $\prec$ node 20, since the string TG occurs for the first time at $S[1]$, while string TA occurs for the first time at $S[4]$. This ordering is different from the more classical approach in which the children of a branch node are ordered lexicographically based on the edge labels from the parent to each child. The advantage of the wotd representation is that the edge labels can be found in constant time, using the left pointer values.

The *wotd* index is implemented as a linear array of 32-bit elements. Note that the term array here refers to the way in which the information about the ST nodes and edges is organized, and should not be confused with the suffix array index, discussed in Section 2.3.2. In *wotd* representation, each branch node occupies two adjacent elements, while a leaf node is recorded using a single element. Figure 6 shows the *wotd* representation of the ST for our running example S = ATGATATGTGAAATAGTAGA$.

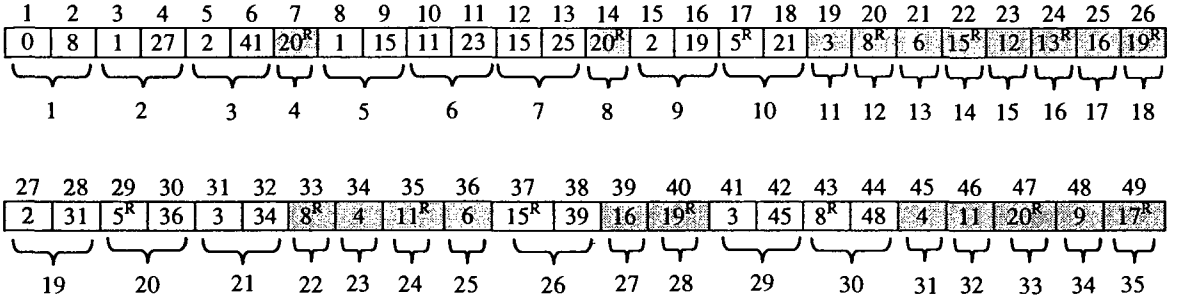


Figure 6. The *wotd* ST Representation of ST for S = ATGATATGTGAAATAGTAGA$

The *wotd* index is a single array of forty nine 32-bit elements, shown in Figure 6 in two parts (first part - elements 1 to 26, second part – elements 27 to 49), due to space limitations. The first row in each part is used to enumerate *wotd* elements and is for illustration purpose only. The ST nodes are evaluated and stored in a top-down, left-to-right manner. Thus, the first two elements represent branch node 1, next two elements represent branch node 2, next two elements represent branch node 3, the seventh element records leaf node 4, etc., as indicated below the figure.

The first of the two 32-bit elements allocated for a branch node u in the *wotd* representation contains the left pointer value $lp(u)$ and two additional bits, called the *rightmost bit* and the *leaf bit*. If the rightmost bit is 1, it indicates that this ST node is the rightmost child of its parent. This is represented in Figure 6 by the superscript R in the corresponding elements. If the leaf bit of a node is 1, it indicates a leaf node; otherwise it

is a branch node. In the second *wotd* element allocated for a branch node u, we store its *firstchild* pointer, which points to the position in the *wotd* index at which the first child of u is stored. For example, in Figure 6, branch node 1 is stored in the first two *wotd* elements, *wotd*[0] and *wotd*[1]. Its first child is branch node 5 (see Figure 5), and hence the value in the second element allocated for node 1 is set to 8, which points to position *wotd*[8], which is the first of the two *wotd* elements that store branch node 5.

A leaf node in *wotd* representation occupies a single element, in which it stores the same information as in the first element allocated for a branch node: the left pointer value, the leaf bit (always set to 1), and the rightmost bit. The leaf nodes in Figure 6 are shown in gray.

The *wotd* ST representation has two main advantages. First, compared to other practical ST representations it is the most space efficient [Giegerich *et al.*, 2003]. In a ST for any sequence with n characters, there are exactly n leaf nodes and up to n -1 branch nodes. Since in the *wotd* representation a leaf node requires 4 bytes and a branch node requires 8 bytes, *wotd* has worst case storage requirement of 12 bytes per character in S. Our experiments show that for real-life DNA and protein sequences, *wotd* requires on average around 9 bytes per character. Second, *wotd* supports efficient exact match search in the memory-based model [Giegerich *et al.*, 2003]. The performance of a *wotd*–based solution is compared with the performance solutions based on linked list (*mccl*) and hash table (*mcch*) representations of the ST [McCreight, 1976], augmented suffix array (*asa*) [Manber and Myers, 1993], and with the time performance of the Boyer-Moore-Horspool (*bmh*) algorithm [Horspool, 1980] for online string searching. The results show that using any ST representation, the search is much faster than the *bmh* algorithm; for example, the

search using *wotd* representation outperforms *bmh* by a factor of 100. Also, the search using *wotd* is faster by an order of magnitude compared to using *asa* representation (basic suffix array plus the *lcp* table, Section 2.3.2). The performance of the search using *wotd*, *mccl*, and *mcch* is comparable, but the latter two representations are not suitable for indexing longer sequences, due to their higher space requirements. For these reasons we find the basic ideas in *wotd* representation to be suitable in our work on indexing biological sequences of various sizes.

However, it has two shortcomings. First, it has a theoretical limit on the input sequence size of around 1 billion bases on 32-bit architectures [Giegerich *et al.*, 2003], but since the *wotd* construction algorithms (reviewed in Section 4.1.) are memory-based, there is a practical limit of around 200 million bases on a regular desktop computer, such as ours with 2 GB RAM. This limits applicability of *wotd* representation for longer sequences which are typical in our context. Second, the experimental evaluation of the exact match search algorithm using *wotd* representation conducted in [Giegerich *et al.*, 2003] was memory-based. The search performance will be significantly affected by disk I/O operations when dealing with large data sequences, whose index cannot fit in memory [Ferragina and Grossi, 1999]. To evaluate the suitability of *wotd* representation for disk-based search, we studied the *wotd* index access patterns during memory-based exact match search. Next section reports our experimental setup and findings.

3.2 *wotd* Access Patterns for Exact Match Search

Performing exact match search (EMS) in a sequence using its memory-resident ST index is a straightforward task, discussed in many textbooks, e.g., [Gusfield, 1997]. The EMS algorithm proceeds in two steps, as follows. Suppose the query is P = AT. In Step 1,

starting from the index root, the algorithm examines the labels of every outgoing edge until a match to the first character (or more) in *P* is found. This process is repeated while traversing the ST downwards, until all the characters in *P* are matched (i.e., *P* is in *S*), or there is no match (*P* is not in *S*). In the former case, the destination node of the last verified edge is marked as the root of the answer subtree, to which we refer as the *answer root*. In our example, the search in the ST from *root* reaches to *node 1* and then to *node 5*, which is the answer root (Figure 5). The subtree at the answer root is referred to as the *answer subtree* since it is the part of the ST that contains all the answers to query *P*. If the answer root is a leaf node, this means there is only a single occurrence of *P* in *S*, which is found. Otherwise, the algorithm proceeds to Step 2, in which it examines the answer subtree where each leaf node represents an occurrence of *P* in *S*. In our example, in Step 2 the algorithm examines the answer subtree rooted at *node 5*, and reaches the leaf nodes *11, 12, 13,* and *14*, which represent occurrences of *P* in *S* starting at locations *S*[0], *S*[5], *S*[3], and *S*[12], respectively (Figure 5).

In [Halachev *et al.*, 2005], we studied the *wotd* index access patterns for performing EMS in the memory-based model. For this, we have implemented a memory-based EMS algorithm, which uses the *wotd* representation. To simulate disk-based EMS search, we considered a fixed size main memory buffer to store currently relevant *wotd* elements. The ST elements are read from disk and placed into this buffer.

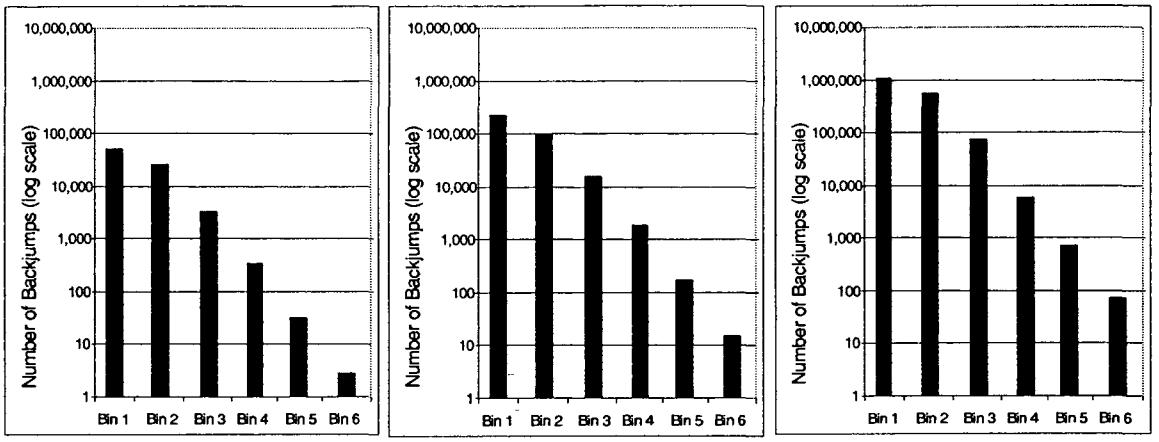
Our goal was to study the locality of reference to *wotd* index for EMS. There are three types of locality of reference: temporal, spatial, and sequential. While the temporal locality is not relevant to traversing the ST nodes when performing EMS, sequential and spatial references are important for two reasons. First, during the traversal of *wotd* by

EMS algorithm, there are cases in which we need to read a ST node, after it has been read previously. We refer to such cases as backjumps. The backjump size is defined as $i - j$ *wotd* elements, where i is the current *wotd* element, j is the next element to which the search refers, and $i > j$. Reducing the backjump size in the disk-based model translates to a higher probability that the element backjumped to is still in the memory buffer, allocated for parts of the ST. Second, good sequential and spatial references allow using simple, well-known double buffering schemes (e.g., pre-fetching) and a simpler buffer replacement (e.g., least recently used) strategy.

In our experiments [Halachev *et al.*, 2005], we used the following 3 real-life sequences as data sequences to be searched: *S1* is chromosome II of *Haloarcula marismortui*, of size 288,050 bases; *S2* is *Homo sapiens* chromosome 7, WGA 47166, contig 47166, of size 1,152,345 bases; and *S3* is the complete genome of *Escherichia coli* O157:H7, of size 5,498,450 bases. These three sequences are DNA sequences over the nucleotide alphabet {A, C, G, T}, obtained in April, 2005 from [NCBI, 2005].

Our results show that in Step 1 of the EMS algorithm the index is accessed in a strictly forward manner, hence there are no backjumps. In order to investigate the index access pattern in Step 2 of the EMS algorithm, we pose the following 4 shortest queries: P_1 = A, P_2 = C, P_3 = G, and P_4 = T against sequences *S1*, *S2*, and *S3* described above. Based on the backjump size, we classified the backjumps into 6 bins, as follows: Bin 1: 0-10 *wotd* elements; Bin 2: 10-10^2 elements; Bin 3: 10^2-10^3 elements; Bin 4: 10^3-10^4 elements; Bin 5: 10^4-10^5 elements; and Bin 6 for backjumps larger than 10^5 elements. Figure 7 shows the average (over the 4 queries) bin distribution observed. Although the majority of the backjumps are of smaller size (i.e., Bin 1 and Bin 2), even for these short

sequences there is significant number of large backjumps (i.e., Bin 5 and Bin 6). These large backjumps will lead to inefficient disk I/O performance and hence slow exact match search in longer sequences, whose index do not fit in memory.



Backjump Distribution for *S1* Backjump Distribution for *S2* Backjump Distribution for *S3*

Figure 7. Backjump Distribution for EMS algorithm using *wotd* representation [Halachev *et al.*, 2005]

In order to reduce the backjump sizes and hence provide a better locality of reference, we proposed an alternative ST representation, in which the ST nodes are evaluated and recorded in the ST in a depth-first, left-to-right order [Halachev *et al.*, 2005]. Under this approach, at the cost of slightly longer index construction times compared to the top-down based *wotd*, the average size of the backjumps for the EMS algorithm is reduced significantly. While the total number of backjumps is the same for the two representations, there is a noticeable shift towards shorter backjump sizes using the depth-first representation.

However, our subsequent work revealed a much more elegant solution for providing better locality of reference to the disk-resident ST index. For ease of discussion, in Figure 5 we showed the starting location of suffixes in *S* by the number below each leaf node, but this information is not stored in *wotd* representation. The reason for having

backjumps in Step 2 of the EMS algorithm is the need to compute the starting location for each leaf node in the answer subtree, which results in additional ST traversals, and eventually backjumps. If this information is available for each leaf node, all the backjumps in Step 2, and for EMS in general, would be eliminated. This is a main idea in our proposed HST representation, discussed in Section 3.5.

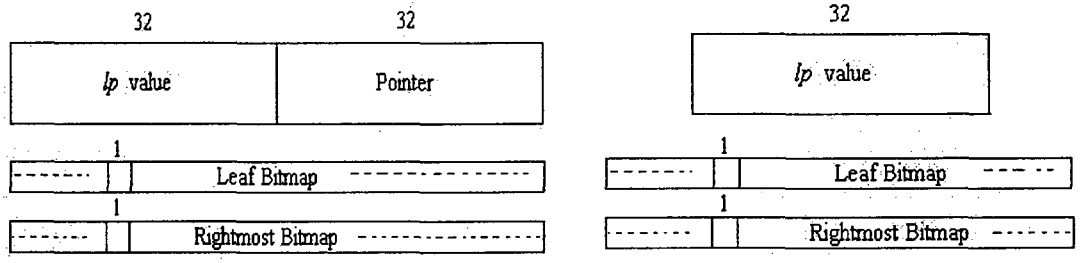
While the proposed solution eliminates backjumps, it does not address the second shortcoming of the *wotd* representation, namely its limit to process sequences of size up to 1GB on 32-bit architectures. To address this issue one possibility is to use the TDD representation, discussed next.

3.3 TDD ST Representation

The TDD representation [Tian *et al.*, 2005] derives its name from the corresponding Top-Down Disk-based ST construction algorithm. Being a variation of the *wotd* representation, it offers a possible solution to its 1GB limit. The TDD representation has a theoretical limit of 2^{32} bases (i.e., sequences up to 4GB sequences) on 32-bit architectures, and in [Tian *et al.*, 2005] it is used to index the entire human genome (of size about 2.7GB) on a typical desktop computer.

The reason for the 1GB theoretical limit *wotd* is that in the first of the two elements allocated to a branch node, as well as in the single element allocated for a leaf node (Figure 6), *wotd* records 3 values - the node's *lp* value, its rightmost bit, and its leaf bit. Since each element in *wotd* representation is 32 bits, only 30 bits are available for recording the *lp* value. The range of the *lp* value is [0, n], where n is the size of the sequence being indexed, in number of characters. Thus, the theoretical limit of *wotd* is 2^{30}, or around 1 billion bases.

Figure 8 shows the TDD representation, which overcomes the 1 GB limit by introducing two additional bitmap arrays – the rightmost bit and the leaf bit data structures, where the rightmost bit and leaf bit for each ST node are recorded. Using this, in the first element for a branch node and in the single element for a leaf node, all 32 bits become available for recording the *lp* value, thus allowing TDD to index sequences of size up to 2^{32} characters, i.e., 4 GB. In summary, the major advantage of TDD ST representation is its ability to index sequences of lengths up to 4GB (most 32 bits operating systems do not support files larger than 4 GB), at the cost of storing the leaf and rightmost bitmap arrays in a separate data structures.



(a) Branch node (b) Leaf node

Figure 8. (a) Branch Node in TDD (b) Leaf Node in TDD

The main focus of [Tian *et al.*, 2005] is on providing an efficient and scalable disk-based ST construction technique (discussed in Section 4.2), which indicated significant improvement compared to existing solutions. However, the subsequent use of the index by various search algorithms did not receive sufficient attention. The proposed solution is not suitable for disk-based model for the same reason as its ancestor, *wotd*, namely the poor locality of reference. Another disadvantage of TDD representation is the necessity to buffer 3 physically separate, but logically related index structures – the main index array (recording the *lp* and pointer values for branch nodes and the *lp* values for leaf nodes) and the two additional rightmost bit and the leaf bit arrays. What makes the

implementation of a buffering strategy even more demanding is that the data in these 3 index structures is of different size. In the main array, each node takes either 4 or 8 bytes (leaf and branch node, respectively), while each element in the two bitmap arrays is of size 1 bit. The exact match and k-mismatch search algorithms available at the authors site [TDD, 2005] are memory-based, i.e., they must read both the sequence and its entire index in main memory in order to perform the search operations. Our experiments using their code for some real-life sequences indicated their poor search time performance, as well as inability to search in some longer sequences (see Section 6.4 for details). In a recent paper, proposed is the Trellis suffix tree representation, which while capable of handling sequences of same size as TDD, addresses some of its search shortcomings. We discuss it next.

3.4 Trellis ST Representation

Same as TDD, the Trellis representation [Phoophakdee and Zaki, 2007] allows for handling sequences of size up to 4GB on a regular 32-bit computer. The Trellis index for a sequence S is not a single file, rather it consists of a collection of suffix trees, each of them for a collection of suffixes from S that share the same prefix. The length of each prefix is computed during index construction and is chosen in such a way that each prefixed suffix tree fully fits in the available RAM memory (given to the construction algorithm as a command line parameter). Further, each prefixed ST consists of two files – one for recording the branch nodes, and one the leaf nodes. Figures 9 (a) and (b) show the structures of branch and leaf nodes in Trellis.

Start Index	End Index	Child $	Child A	Child C	Child G	Child T

(a) Branch Node

Start Index	Suffix Index

(b) Leaf node

Figure 9. (a) Branch Node in Trellis (b) Leaf Node in Trellis

Each branch node occupies seven 4 byte elements, for a total of 28 bytes. The first two elements represent the edge between this branch node and its parent, determined by *S*[Start Index, End Index]. The next 5 elements allocated for a branch node represent this branch node's outgoing edges. In each child element, recorded is a numerical value x, which indicates if an outgoing edge starting with the corresponding symbol does not exist ($x = 0$), or in case it exists, if the outgoing edge leads to a branch ($x >$ *threshold*) or a leaf node ($x <$ *threshold*).

Each leaf node occupies two 4 byte elements, for a total of 8 bytes. The first element represents the edge between this leaf node and its parent, determined by *S*[Start Index, End of Input String]. This leaf corresponds to the (Suffix Index)-th suffix of *S*.

The Trellis representation offers three advantages compared to TDD. First, using variable prefix length, each prefixed suffix tree is constructed in a memory-based manner, thus reducing the disk I/O overhead. As a result, Trellis exhibits faster the index construction times compared to TDD in general. Second, using the information stored in the child elements for a branch node in Trellis allows for more efficient disk-based traversal of the ST. As a result, the exact match search using the Trellis index is faster compared to the memory-based TDD search. Third, Trellis provides the option after completing the ST index construction, to compute and record the suffix links. Suffix links were introduced in order to speed up the ST construction and ST traversal in some search problems. The notion of suffix link is described as follows [Gusfield, 1997]. Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly

empty) substring. For an internal node v in the ST with path-label $x\alpha$, if there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a *suffix link*. Since the search applications considered in this work do not require the availability of the suffix links, for the fairness of comparison we do compute and record them for Trellis.

However, Trellis has two shortcomings. First, its index size is large and is proportionate to the alphabet size (Figure 9.a). Even for the small, five symbol DNA alphabet (i.e., A, C, G, T, \$), the size of the Trellis index for real-life sequences is on average $28n$ bytes (up to $50n$ bytes, if suffix links are recorded), where n is the number of characters in S. In comparison, *wotd* requires $9n$ bytes and TDD requires $13n$ on average. For this reason, Trellis is suitable for DNA sequences only. Second, similar to *wotd* and TDD, Trellis does not address the issue of the poor locality of reference during disk-based search.

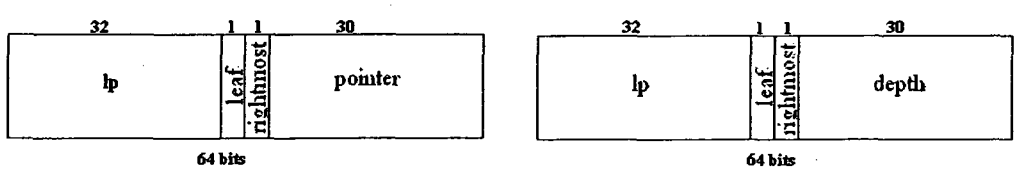
In our design of ST representation for biological sequences, we focused on solution which will be capable of indexing long sequences, can be used for both DNA and protein data, and will provide good locality of reference to search applications. Our proposed representation, HST, handles sequences up to 4GB (as for TDD and Trellis), and is suitable for disk-based search in DNA and protein sequences, as discussed next.

3.5 Our Proposed HST Index

We propose a two-level index structure, called HST (for Halachev, Shiri, Thamildurai), which combines a lookup table (LT) and a suffix tree (STTD64). The small, memory-resident LT serves as an index to the large, disk-resident STTD64. Next, we describe these two data structures.

We proposed the STTD64 (Suffix Tree, Top-Down, 64 bits) representation of the ST in [Halachev *et al.*, 2007]. Each STTD64 node is represented as a single 64-bit record, regardless of being a branch or a leaf node. The STTD64 index for a sequence S is implemented as a linear array of these records. Note that our design choice for using 64 bits for representing the ST nodes does not imply in any way a 64-bit architecture, although it would benefit from it. All our implementations and experiments are done using a standard 32-bit machine.

Figures 10 (a) and (b) show the structures of branch and leaf nodes in STTD64. For both type of nodes, the first 32 bits are allocated for storing the *lp* value, thus overcoming the 1 GB limit of *wotd*. Note that the *lp* value is bounded by the size of the sequence, n, thus it can be represented using only $\lceil \lg n \rceil$ bits. In our work, we adopt a uniform node design, i.e., regardless of the sequence size 32 bits are reserved for the *lp* value, in order to be able to handle sequences up to 4GB. This uniform design makes node processing much simpler and more efficient. Bit 33 records the leaf bit value and bit 34 records the rightmost bit value. For a branch node, the remaining 30 bits are used to store the pointer to its first child. The main difference between *wotd* (and all existing ST representations known to us) and our STTD64 index is that we use the last 30 bits of a leaf node store its *depth*.



(a) Branch node (b) Leaf node

Figure 10. (a) Branch Node in STTD64 (b) Leaf Node in STTD64

Recall that the depth of a leaf node v is defined as the number of characters on the path from the root of the ST to the parent of v. As we will explain shortly, by storing the

depth value for each leaf node, there is no need to traverse the ST for computing the starting positions of suffixes, and thus all backjumps are eliminated.

The other component of HST, the LT index, is similar to the bucket table for suffix arrays proposed by [Manber and Myers, 1993], and later adapted in various forms in [Baeza-Yates *et al.*, 1996], [Abouelhoda *et al.*, 2004], [Phoophakdee and Zaki, 2007], and [Sinha *et al.*, 2008]. Our LT index contains the following information. Given the sequence alphabet Σ and a small fixed parameter p, consider all $|\Sigma|^p$ lexicographically sorted unique strings over Σ with length p, i.e., l_0, l_1, ..., $l_{|\Sigma|^p-1}$. In the LT index, implemented as an array of pointers to STTD64 nodes, for each l_i we store the pointer to the STTD64 node u_i whose edge-labels from the root to u_i spell out exactly l_i. If such a node u_i does not exist (e.g., l_i is not in S), we set $LT[i]$ to NULL. One advantage provided by the LT index is: given a query P of size m symbols, we can use the memory-based LT index only to process the first p symbols, thus reducing the overall number of disk I/Os for P. The availability of the LT index allows for another important improvement of the locality of reference to the disk-resident STTD64, as will be explained in Section 5.1. Figure 11 shows the HST index, i.e., the STTD64 index and the LT index (with $p = 2$), for our sample sequence S = ATGATATGTGAAATAGTAGA\$.

The STTD64 nodes are evaluated and recorded following the top-down, left-to-right order, illustrated in the figure by the numbers on the top row. For example, the first record represents branch node 1 (in Figure 5), the next record represents branch node 2, the third record represents branch node 3, etc.

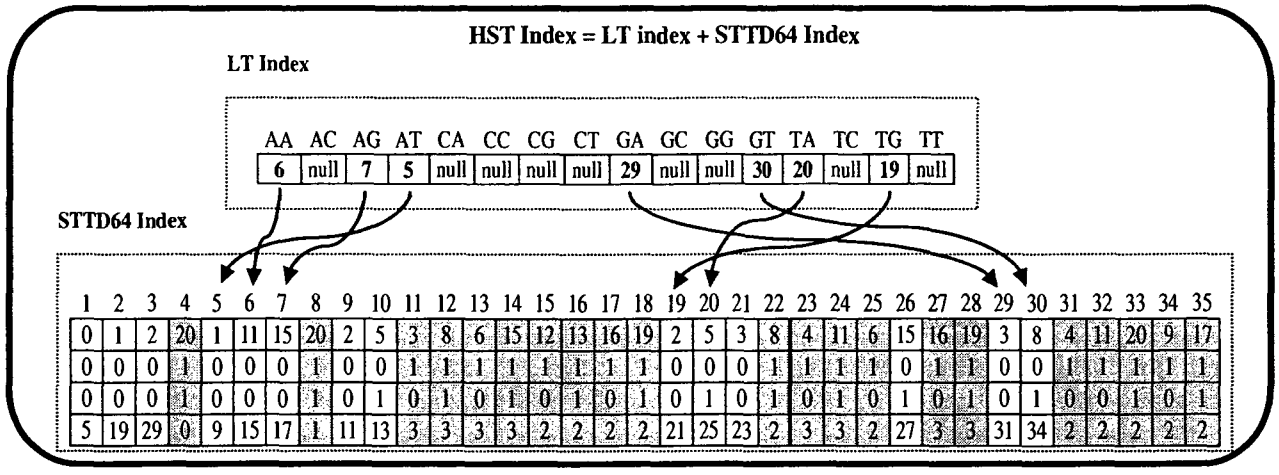


Figure 11. HST Index for S = ATGATATGTGAAATAGTAGA$

In the figure, each STTD64 node is represented vertically as a 64-bit record, divided into four fields – *lp* value, leaf bit, rightmost bit, and pointer/depth. For clarity, leaf nodes are shown in gray in the figure. To illustrate the information in the LT index, consider for example LT[0]. In LT[0], we store value 6, i.e., the pointer to STTD64[6], which is the node that represents the substring "AA". Note that the labels in the first row of the LT index are for illustration only. Given a substring l_i, its location in the LT index is calculated using the order A < C < G < T.

Our proposed HST index overcomes the 1GB limit of the *wotd* representation. HST can handle longer sequences, with a theoretical limit of 4 GB, thus matching the capabilities of TDD and Trellis. In [Halachev *et al.*, 2009], we have shown HST effectively constructed in 16 hours for the entire human genome (approximately 2.7 GB) on a regular desktop computer with 2 GB RAM. Since the depth of a leaf node v is equal to the size of the prefix that the suffix ending in v shares with some other suffix, the 30 bits allocated for recording the depth values of leaf nodes introduce a restriction on the maximum size of the repeats contained in the sequence. Theoretically, this means HST

cannot handle sequences containing exactly repeated substrings of size longer than 2^{30} (1 billion) bases. However, this restriction is not likely to be an issue when dealing with real-life biological data.

This advantage of HST over *wotd* comes at the cost of increased storage requirements. The number of nodes in *wotd* and STTD64 suffix tree representations is equal and is at most $2n$, where n is the length of the sequence being indexed. Since each ST node is 8 bytes in our representation, the STTD64 index requires up to $16n$ bytes in the worst case, compared to $12n$, required by *wotd*, the most space efficient, uncompressed ST representation. Our experiments show that for real-life DNA and protein sequences, STTD64 requires about $13n$ bytes on average, compared to $9n$ for *wotd*. The size of the LT index is independent of the sequence size and is $4|\Sigma|^p$ bytes. For DNA sequences and for $p = 7$, for example, size of the LT index is only 64KB. The HST index size is similar to the one for TDD and half of the Trellis index size. Considering that external memory is easily affordable, we believe that the additional $4n$ bytes required by HST compared to *wotd* is a reasonable cost, noting the longer sequences HST can handle and its suitability for the disk-based model.

The second advantage of HST over *wotd*, TDD, and Trellis is the improved locality of reference to the disk-resident index. The availability of the leaf depth values in STTD64 significantly improves locality of reference by eliminating backjumps in Step 2 of exact match search. For search applications which use HST, depth values allow for fast computation of the starting locations of the suffixes. For example, the start location of the suffix encoded by the path from the root to node 17 is computed by subtracting the depth value of node 17 from its *lp* value, i.e., the start location of the suffix "AGTAGA$" is 16

$-2 = S[14]$ (see Figures 5 and 11). Storing the information needed for this computation in a single node eliminates the ST traversals required by *wotd*, TDD, and Trellis based solutions, leading to significant decrease in the number of disk I/O operations, and eventually to improved search times for applications based on HST. The advantages of our HST design become more evident when we consider HST based search algorithms (Chapter 5) and experimental results (Chapter 6). We present the HST index construction algorithm in the following Chapter, in which we also review existing ST construction algorithms and highlight their advantages and shortcomings.

4 Suffix Tree Construction Algorithms

As already discussed, the major problem in using ST is the index size being large, which requires having either large main memory or efficient disk I/O construction and search algorithms. Recall that we refer to indexing techniques which require both the sequence and its index to be in the main memory as *memory-based*, and to techniques which access the disk to read or write related parts of the index as *disk-based*.

While it is possible to use a memory-based approach to construct ST for shorter sequences, longer sequences may impose impractical requirements on the amount of main memory needed. For example, it is estimated that a memory-based construction of the ST index for the entire human genome of size approximately 3 GB would require about 45 GB of RAM [Kurtz, 1999]. In our work, we develop an efficient disk-based ST index construction technique, suitable for sequences of various sizes due to its efficient use of available resources. Next, we review existing ST construction algorithms and highlight their contributions.

4.1 Memory-Based ST Construction Algorithms

Until recently, majority of ST construction algorithms proposed were memory-based, i.e., the ST was constructed fully in memory. The first algorithm for building suffix trees is due to Weiner [Weiner, 1973], where the suffix tree is called a Position Tree. A more space efficient algorithm is proposed in [McCreight, 1976]. The most popular solution is provided by Ukkonen [Ukkonen, 1995], which is a variant of [McCreight, 1976]. Ukkonen's solution allows easier proof of bounds and is easier to implement. These three algorithms share some properties. First, they are memory-based, i.e., they require both the

sequence and its ST index to fit and reside in the main memory. Second, they all compute and use suffix links. Third, using the suffix links in the construction phase, these algorithms build the ST in time linear to the size of the sequence. While these techniques are suitable for short sequences, they are limited by the size of the available main memory, which in turn limits their applicability. For longer sequences, which are typical in the bioinformatics domain, constructing and using a ST index dictates disk utilization. In this case, numerous disk I/O operations are performed during construction and search tasks. Thus, for a disk-based technique to be successful, good locality of reference and efficient buffer management strategies are crucial to have.

The issue of locality of reference during ST construction is addressed in [Giegerich *et al.*, 2003]. In conjunction with their *wotd* ST representation (Section 3.1), the authors proposed two ST construction algorithms - *wotdlazy* and *wotdeager*. The first technique is used to construct only those parts of a ST which are related to a particular query, while the second technique builds the entire suffix tree, thus being more relevant to our work. Both algorithms are memory-based, which limits their applicability on typical desktop computers (with 2 GB RAM) to sequences of size about 200 million bases for *wotdlazy* and even less for *wotdeager*. Further, since *wotdlazy* and *wotdeager* algorithms do not compute, use, and record suffix links, their time complexity is no longer linear; instead they are $O(n \log n)$ for the average case, and $O(n^2)$ for the worst case. These two algorithms have one significant advantage: while not using suffix links leads to worse time complexity, it provides better locality of reference, i.e., the ST is accessed in sequential manner. This trade-off was adopted by some subsequent works on providing efficient and scalable disk-based ST construction techniques, as discussed next.

4.2 Disk-Based ST Construction Algorithms

Recent studies [Hunt *et al.*, 2002; Bedathur and Haritsa, 2004; Cheung *et al.*, 2005; Tian *et al.*, 2005; Phoophakdee and Zaki, 2007] focused on proposing efficient disk-based ST construction algorithms, by relaxing the requirement that both the sequence and the index must fit fully in the main memory. Instead, they assume the data sequence is in main memory and construct the ST by utilizing secondary storage, i.e., the disk.

In [Bedathur and Haritsa, 2004], the authors investigate the problem of developing an efficient buffer management strategy for extending memory-based ST construction algorithms and propose TOP-Q, which augments Ukkonen's algorithm to disk-based model. The longest sequence indexed using their technique is reported to be of size 70 MB, and the construction took around 14 hours.

In [Hunt *et al.*, 2002], the authors observe that although suffix links provide a linear ST construction time, their usage leads to poor locality of reference, which is a major bottleneck for extending existing memory-based algorithms (such as Ukkonen's) to disk-based model. The authors implement two solution ideas: abandonment of suffix links and partitioning the sequence to be indexed into smaller sequences, for which the ST can be built in-memory. This way, the ST is constructed for each partition (independent of other partitions) and checkpointed to the disk, thus freeing the main memory to process next partition. As the algorithm avoids suffix links, its time complexity is $O(n \log n)$, but allows building a ST for sequence of size 263 MB, done in around 19 hours.

In [Cheung *et al.*, 2005], the authors propose DynaCluster, a ST construction technique based on dynamic clustering of the suffixes. The ST is constructed one cluster at a time, using only a small amount of memory for each cluster. DynaCluster also

abandons the computation and use of suffix links during construction; as an additional option they can be computed after the ST is constructed. The time complexity of DynaCluster is $O(n \log n)$ on average, $O(n^2)$ in the worst case. It exhibits faster index construction times compared to the solutions in [Bedathur and Haritsa, 2004] and [Hunt *et al.*, 2002]. For example, for a sequence of size 100 MB, it takes around 20 minutes for DynaCluster to construct the ST index, including the additional suffix link computation. However, it is shown to be restricted to sequences up to 200 MB [Phoophakdee and Zaki, 2007].

In [Tian *et al.*, 2005], the authors propose an extension of the ideas in [Hunt *et al.*, 2002; Giegerich *et al.*, 2003; Bedathur and Haritsa, 2004]. They suggest a Top-Down, Disk-based (TDD) ST construction technique, which abandons suffix links, partitions the input sequence, and employs efficient buffering strategies to create the TDD index (Section 3.3). The suffixes of the sequence to be indexed are grouped in several partitions, based on the common prefix they share. For each partition, they use *wotdeager* algorithm [Giegerich *et al.*, 2003] to build the corresponding part of the suffix tree. Since even with partitioning, some of the partitions may not fit in the main memory, another contribution of [Tian *et al.*, 2005] is a buffering strategy for all required data structures. The time complexity of the construction algorithm is $O(n \log_{|\Sigma|} n)$ on average, and $O(n^2)$ in the worst case. The TDD technique was the first to construct the suffix tree for the entire human genome (about 2.7GB), and this was done in 30 hours on a typical desktop computer with 2 GB main memory and a single processor. For shorter sequences which could be handled using Hunt's solution [Hunt *et al.*, 2002], the construction of a TDD index is about 7 times faster [Tian *et al.*, 2005]. Further, in practice TDD also

outperforms the linear memory-based Ukkonen's solution, due to the improved locality of reference and taking advantage of caching opportunities.

In a recent work, [Phoophakdee and Zaki, 2007] propose Trellis, the most efficient ST index construction technique up-to-date. It constructs the ST index for a sequence in several steps. Initially, it partitions the sequence - first to shorter substrings and next using variable prefix lengths for the suffixes in each substring. As a second step, for each such set of suffixes that share the same prefix, its *prefixed ST* is constructed fully in memory, and upon completion written to the disk. In the third step, the prefixed STs from different substring partitions which share the same prefix are merged. The last step, which is optional, is to compute and record the suffix links. The experimental results in [Phoophakdee and Zaki, 2007] show that Trellis constructs the suffix tree for the entire human genome in 4 hours, and can compute the full set of the suffix links within an additional 2 hours. Although its time complexity is $O(n^2)$ in the worst case, i.e., same as TDD, due to prefixed ST being constructed in-memory and thus reducing the disk I/O overhead, Trellis is 2 to 4 times faster than TDD in index construction. Compared to other ST construction techniques which compute the suffix links, Trellis is several orders of magnitude faster [Phoophakdee and Zaki, 2007].

Next, we describe the construction technique of our proposed HST index.

4.3 HST Index Construction

Our proposed disk-based HST index construction technique [Halachev *et al.*, 2009] is an extension of TDD [Tian *et al.*, 2005], which in turn extends *wotdeager* technique [Giegerich *et al.*, 2003] by adding a partitioning step and introducing a suitable buffering strategy. The HST and TDD techniques have similar features, including: (*i*) abandonment

of suffix links, (*ii*) partitioning of the input string using a fixed prefix length, (*iii*) utilization of *wotdeager* algorithm for each partition, and (*iv*) employing buffering techniques, which make good use of available main memory and cache. The HST construction algorithm is shown in Figure 12 and was primarily developed by Anand Thamildurai [Thamildurai, 2007].

Algorithm *constructHST* (Sequence *S*, int *prefixlen*, int *p*){
Phase1: Partition *S*
1. Scan the Sequence *S* and partition *Suffixes*
   based on the first *prefixlen* symbols of each suffix

Phase2: For each partition, do the following:
2. Populate *Suffixes* from current partition
3. Sort *Suffixes* on the first unevaluated symbol
4. Output branch and leaf nodes to *STTD64*
5. Push the branch nodes into the *Stack*
6. while (*Stack* is not empty)
7. Pop a branch node
8. Find the Longest Common Prefix (LCP) of all the suffixes in this node's range by checking *S*
9. Sort the range in *Suffixes* on the first unevaluated symbol
10. Output branch and leaf nodes to *STTD64*
11. Push the branch nodes into the *Stack*
12. end while

Phase3: Depth Filling
13. Traverse *STTD64* by following the branch pointers
    and compute and record the depth value for each leaf node

Phase4: LT index construction
14. For each unique substring of size *p* characters,
    using *STTD64* obtain and record the LT values
} // **end *constructHST***

Figure 12. HST Construction Algorithm

The construction process proceeds in four phases. In the first phase, the suffixes are partitioned based on the first prefix length (*prefixlen*) characters and recorded in the corresponding *Suffixes* data structure. Partitioning allows generating smaller, disjoint parts of the suffix tree which can be managed and built independently of each other. This

reduces the main memory requirements of the algorithm, noting that the cost of partitioning increases linearly with *prefixlen*.

In the second phase, for each partition we do the following. For every branch node, we evaluate all its children nodes and pick the *firstchild* (w.r.t. $\prec$) to be evaluated next. For example, consider the ST in Figure 5. HST construction algorithm first evaluates the children of the root, i.e., nodes 1, 2, 3, and 4 (w.r.t. $\prec$). All branch nodes (1, 2, and 3) are written to the *Tree* and also pushed into the *Stack*, in order to process their children nodes later. The leaf node 4 is written only to the *Tree*. In the next iteration, node 1 is popped from the *Stack* and its children are evaluated. The processing of other nodes continues in the same manner.

While TDD and HST construction algorithms share Phases 1 and 2 (Figure 12), the additional Phases 3 and 4 are specific to HST construction. In Phase 3 of HST, computed and recorded are the *depth* values for all leaf nodes (Figure 10.b). The availability of this information plays a crucial role during search using HST index, as will be shown later. This additional phase in HST requires at most $O(n^2)$ constant time operations.

The other additional phase of HST compared to TDD is the construction of the LT index. The LT index can be constructed either in conjunction with STTD64 index construction or by traversing the STTD64 index for each substring l_i after STTD64 is constructed. We take the latter approach for its simplicity and independence from the actual STTD64 construction algorithm. Hence, Phase 4 requires $O(p|\Sigma|^p)$ time for traversing the STTD64 index from the root to the last character in each l_i string. For real-life DNA sequences and practical p values, the term $|\Sigma|^p$ is usually much smaller than n.

Thus, the overall time complexity of our HST construction algorithm remains $O(n^2)$, as is for TDD.

Even though TDD and HST construction algorithms have the same time complexity, their practical performance depends on efficient use of the available main memory, as well as on the nature and the amount of disk I/O operations. In [Tian *et al.*, 2005] different buffering techniques have been proposed and evaluated. Each of the four memory resident data structures – *Text*, *Suffix*, *Temp*, and *Tree* use buffering strategies based on the nature of their disk I/Os. Even though buffering improves the scalability with respect to the sequence size, it degrades performance of the algorithm compared to a memory-based technique. It is then preferable to keep as much data as possible in the main memory. In TDD construction algorithm the sizes of these four buffers are chosen at the beginning and fixed during the ST construction process. Thus, even when the required data for particular partition iteration could be accommodated in memory, it is buffered. In contrast, in our implementation of the HST index construction algorithm, we use a dynamic buffering scheme during construction of the STTD64 index, which adapts the four buffer sizes considering the size of each partition. This dynamic buffering is the second major difference between TDD and HST construction algorithms, and it leads to improved disk I/O performance of the HST construction algorithm, which compensates for the additional phase of computing and recording the depth values, as shown by our experiments. Our results in Section 6.2 indicate that TDD and HST techniques exhibit comparable ST index construction performance for sequences longer than 100 MB. More details on the HST construction algorithm can be found in [Halachev *et al.*, 2009; Thamildurai, 2007; Halachev *et al.*, 2005].

Although the construction time and storage space are important criteria for comparing indexing techniques, the efficiency of the search applications using the indexes is very important in practice, because index creation is done only once, while searches using the index occur very many times. In the next chapter, we present a set of search applications which use the HST index.

5 Search Applications Using HST

The purpose of any indexing technique is to support efficient, scalable, and versatile search tasks on the indexed data. In our work we focused on three search problems – exact match search, k-mismatch (approximate) search, and structured motif search. Exact match search is at the core of many search applications on sequence data. In other words, if an index representation does not provide efficient exact match search, there is little hope that it provides efficient support for more involved search tasks. The k-mismatch problem illustrates the performance of HST for one type of approximate search, which has practical standalone applications, as well as serving as a core component in more advanced searches. The structured motif search solution based on HST supports another practical bioinformatics application. Yet another application currently supported by HST index is supermaximal repeat search [Lian *et al.*, 2008].

One of the challenges we faced in development of search applications based on HST was the fact that most of the previously known ST based search algorithms rely heavily on suffix links for efficiency, e.g., [Chang and Lawler, 1994; Gusfield, 1997; Höhl *et al.*, 2002; Delcher *et al.*, 2002; Bray *et al.*, 2003; Gusfield and Stoye, 2004; Carvalho *et al.*, 2004]. Since our HST index does not record suffix links, we answer an important question: How practical is a ST index that does not contain suffix links?

Next, we present our HST based algorithms to exact match search, k-mismatch search, and structured motif search.

5.1 Exact Match Search Algorithm (STEM)

As mentioned above, exact match search (EMS) is at the core of numerous exact and approximate search applications in bioinformatics, including finding different types of repeats and palindromes, searching for motifs, etc. Also, EMS is used as the first step in similarity search tools for biological sequences, such as BLAST [Altschul *et al.*, 1997]. Searching for similar DNA sequences using the standalone BLASTn is a time-consuming task (Section 1.4), and 85% of its overall search time is spent performing EMS [Cameron, 2006]. Thus, a more efficient solution to EMS is desired.

The EMS problem is defined as follows. Given a query pattern P with m characters, the problem is to find all exact occurrences of P in data sequence S with n characters, $m<<n$, and report their starting locations. Our disk-based exact match search algorithm, called STEM (for Suffix Tree Exact Match), proposed in [Halachev *et al.*, 2009], is shown in Figure 13. It uses the memory-based solution [Gusfield, 1997] explained in Section 3.2., and extends it with adding a buffering strategy (Figure 13, Phase 1). The HST index representation provides three sources for disk I/O improvement, discussed next.

First, the LT index eliminates the need to traverse the STTD64 index for the first p characters of each query P (lines 5 and 16), thus reducing the number of disk I/Os performed in Phase 2.-The exact match search algorithms based on SPAT [Baeza-Yates *et al.*, 1996], ESA [Abouelhoda *et al.*, 2004], Trellis [Phoophakdee and Zaki, 2007], and LOF-SA [Sinha *et al.*, 2008] also take this advantage.

```
Algorithm STEM (Sequence S, Index HST, Query Set QS){
Phase1: Buffering Strategy
1. Get S, LT, and QS from disk
2. Sort the queries in QS in lexicographical order
3. oldNode = NULL;
4. for each query P in the sorted QS
5.   use LT to determine firstNode for P (use first p chars of P)
6.   if (firstNode == NULL) //the p-prefix of P not in S
7.     return (P is not in S);
8.   else if (firstNode == oldNode)
9.     continue; //same STTD64 subtree needed
10.  else //reset Buffer (contains STTD64 pages read so far)
11.    discard Buffer content;
12.    use LT to determine lastNode in the STTD64 subtree;
13.    Buffer size = lastNode - firstNode;
14.    Allocate memory for Buffer;
15.    set oldNode = firstNode;
     } //end setting the Buffer for query P

// Conduct the exact match search in two phases (following [Gusfield, 1997]),
// If page not in memory, get it from disk, keep it in Buffer
Phase2: Find the ans_root for query P
16. ans_root = firstNode; p_pntr = p;
17. while (p_pntr < m) //there are unexamined characters in P
18.   find ans_root's outgoing edge label to match P[p_pntr];
19.   if (there is no such edge)
20.     return (P is not in S);
21.   else // i.e., there exist such an edge e
22.     ans_root = destination node of edge e;
23.     p_pntr = p_pntr+|e|; //|e| is the number of symbols in e
24. } //end while, at this point all P chars are matched
Phase3: Given ans_root, reach each leaf node in its subtree
25. if (ans_root is leaf node x)
26.   return (Single Occurrence of P in S starting at S[lp(x)-depth(x)]);
27. else //process the answer subtree rooted at ans_root
28.   for each leaf node y in the answer subtree
29.     return (P occurs in S starting at S[lp(y)-depth(y)])
30.   }
31. } //all occurrences of P in S are found and reported
32. }//end for each query P in QS
} // end STEM
```

Figure 13. STEM Search Algorithm

Second, the availability of the depth value for each leaf node in STTD64 representation significantly improves the locality of reference to the index in Phase 3, by avoiding the traversal of the answer subtree from its answer root to each of its leaf nodes.

This process is required for the suffix tree based search using TDD [Tian *et al.*, 2005] and Trellis [Phoophakdee and Zaki, 2007], in order to compute the starting locations of P in S. As shown in Section 3.2., these ST traversals would lead to many disk I/Os with poor locality of reference and would slow down the search process. Instead, once the answer root is determined in Phase 2, STEM treats the nodes in the answer subtree as a flat array of STTD64 elements, and accesses them strictly sequentially, from left to right. Each leaf node in this subtree represents an occurrence of P in S (line 29) and its starting location is computed by subtracting the depth value of the leaf node from its *lp* value. This leads to a significant improvement in the number and nature of disk I/Os for the search.

Last, in bioinformatics applications it is often the case that S is searched for a *set of queries*, rather than a single query. For example, in BLAST suite of programs, BLASTn looks for sequences similar to a query sequence Q, by first generating the set QS of all m-substrings of Q, called *words* in BLAST terminology. The set contains $|Q|-m+1$ queries of length m. Performing sequential scans of the database, BLASTn then looks for exact matches for all the words in the query set in each database sequence, a step which takes up to 85% of total BLASTn time [Cameron, 2006].

Existing ST-based techniques do not adapt well for such applications. During exact match search, for each requested ST node, Trellis reads into memory only the index page containing it. If the requested node is not already in memory, this page is discarded and the required page is read from disk. While this solution has the advantage of small RAM requirements, for a set of queries it results in repeated reads of some ST pages. TDD avoids these repeated readings by allocating memory for the entire ST index. Thus, only the pages needed are read, and they are read exactly once for the entire query set. The

drawback of this approach is the considerable RAM requirements, which on desktop computers with 2GB RAM limit the TDD-based exact match search algorithm to sequences shorter than 100MB.

In contrast, the HST index allows for an efficient and scalable buffering scheme, shown in Figure 13 (Phase 1), which is the third source of disk I/O improvement. It guarantees that for a set of queries no STTD64 page is read from disk more than once, while supporting exact match search in long sequences (e.g., 2.7GB) with only 2GB RAM. To achieve this desirable characteristic, we use the LT index. Note that the information about all suffixes in S that share the same l_i prefix (of size p characters) is represented in the subtree rooted in STTD64[$LT[i]$]. Thus, any query P starting with the same l_i prefix will never access STTD64 index nodes outside this subtree. Our STEM technique sorts the queries P in the query set QS (line 2) and allocates in RAM the space needed for each such subtree (line 14), whose size is significantly smaller than the size of the entire STTD64 index. Thus, if two or more subsequent queries share the same prefix of size at least p, the STTD64 index nodes read for the first query and kept in *Buffer* are available to be reused by subsequent queries.

Next, we provide an analysis of the disk I/O operations performed by STEM. For this, we use the usual two-level memory model, as in [Ferragina and Grossi, 1999], which assumes we have a small but fast main memory (i.e., RAM) and a large but slow external memory (i.e., HDD). The external memory is partitioned into disk pages, each of size B bytes. The smallest unit of information that can be read from or written to external memory is one disk page, and each such operation is referred to as a *disk access*. In our analysis, we count a *new disk access* every time the currently accessed disk page is

different from the last one accessed. This simplification does not consider the probability that some requested disk pages are already read from disk and available in the Buffer (Figure 13, Phase 1). Thus, our results give an upper bound for the total number of disk pages read from disk, which allows for a more general analysis of STEM disk I/O, independent of main memory size, on one hand, and the number, the length, and the content of the queries in the query set, on the other.

In Phase 2, STEM requires access to both the sequence and its HST index, while in Phase 3, the search continues by accessing the STTD64 index only. Our experiments indicated that for searching in a given sequence for a set of EMS queries, it is faster to read the whole sequence from the disk to main memory, compared to buffering it. Thus, as an initial step, STEM reads the entire sequence (of size n bytes, assuming 1 byte per character) from the disk, which requires $\lceil n/B \rceil$ disk accesses. We also read fully into memory the LT index and the query set QS, but their sizes are usually negligible compared to the size of S and its STTD64 index, hence we do not consider them in our analysis.

However, for the STTD64 index, which is an order of magnitude larger than the sequence, this read-all strategy may not be applicable or even desirable. Instead, in our analysis we assume that STEM reads the needed ST information from the disk using a single buffer of size B bytes (i.e., one disk page).

During the downwards traversal of the HST in Phase 2, STEM performs at most $(|\Sigma|*8)/B$ disk accesses per each ST node examined, where $|\Sigma|$ is the sequence alphabet size and 8 bytes is the size of our index node. For disk page size of 4KB and alphabet size less than 512 symbols, the term $(|\Sigma|*8)/B$ translates to at most 1 disk access per ST node.

Since the number of the ST nodes on the path from the ST root to the answer root is at most equal to the number of characters in P, in Phase 2 STEM performs at most $m - p$ *random* disk accesses per query P, where p is the prefix length used in the LT index.

In Phase 3, since the depth values are available, the answer subtree for a query P is processed as a flat array of cells, thus avoiding unnecessary and costly ST traversals. Since each leaf node in the answer subtree corresponds to an occurrence of P in S, all leaf nodes in the answer subtree must be visited. Let *occ* denote the number of leaf nodes in the subtree, which also indicates the number of occurrences of P in S. Since the number of branch nodes in any subtree of a ST is at most equal to the number of leaf nodes in that subtree, we access at most $2*occ$ index nodes in Phase 3 while processing sequentially the answer subtree. Considering the STTD64 node size of 8 bytes, STEM performs at most $(2*8*occ)/B$ *sequential* disk accesses per query P in Phase 3. Thus, our STEM algorithm requires at most $O(n/B + (m - p) + occ/B)$ disk accesses per query P, where only the second term represent random disk accesses, while the other two terms are strictly sequential access to the disk.

In comparison, the optimal (in terms of disk I/O) exact match solution for a single query is based on the String B-Tree index [Ferragina and Grossi, 1999] and requires $O(\log_b n + m/b + occ/b)$ disk accesses per query, where b is the number of *atomic* elements (such as integers, characters, and pointers) in one disk page. Since [Ferragina and Grossi, 1999] do not present an actual implementation of the String B-Tree data structure, to simplify the comparison, we assume $O(b) = B$, i.e., each Sting B-Tree atomic element is of size only 1 byte, which is in favor of String B-Tree. Thus, performing exact match search using String B-Tree requires $O(\log_B n + m/B + occ/B)$ disk accesses per

query. We compare the two techniques for a more practical scenario, in which the index is used to search for a set of x queries. In this case, STEM will perform $O(n/B + x(m - p) + x(occ/B))$ disk accesses, since sequence S is read only once in the initial step, while String B-Tree requires $O(x(\log_B n + m/B + occ/B))$ disk accesses. Note that the last terms for each technique are the same and that m is usually small. The main difference between the two techniques is their first terms, n/B and $x\log_B n$, respectively. For smaller query sets, where $x < n/(B\log_B n)$, String B-Tree will have an advantage over STEM with respect to the number of disk accesses However, for larger query sets, where $x \geq n/(B\log_B n)$, STEM will perform less disk accesses than String B-Tree. For example, searching in sequence of size n = 100MB (e.g., 100 million bases, the average size of a human chromosome) using a typical page size B = 4KB, the threshold value for x is about 1700, i.e., if the query set contains more than 1700 queries, it can be expected that STEM will have an advantage over String B-Tree. This indicates that for practical number of queries in a query set, STEM exhibits acceptable disk I/O performance, comparable to the optimal solution for a single query, provided by String B-Tree.

The time complexity of STEM is determined as follows. First, to determine the LT element corresponding to the first p characters, we perform $\Theta(p)$ constant time calculations. Next, in Phase 2, the STTD64 index is traversed for the remaining $m - p$ query characters. In the worst case, each edge on the path from the ST root to the answer root will be of length 1 character. Thus, there are at most $m - p$ ST nodes, whose set of outgoing edges have to be examined looking for a match for each P[p_pntr] character (Figure 13, Step 18). This leads to at most $O(|\Sigma|(m - p))$ characters comparisons, where $|\Sigma|$ is the alphabet size. Last, in Phase 3, the answer subtree is processed to calculate the

starting position of all *occ* occurrences, represented by the leaf nodes in this subtree. As already explained, in any subtree the number of branch nodes is at most equal to the number of leaf nodes, hence this step takes at most $O(2*occ)$ constant time calculations. Thus, the time complexity of the STEM algorithm is $O(m + occ)$ in the worst case.

5.2 k-mismatch Search Algorithm (STKM)

While exact match search has its own practical importance, some other bioinformatics applications require more flexible search tasks. Another search task that we implemented is the k-mismatch search, which is an approximate search problem used in bioinformatics to account for possible sequencing errors, as well as for some simple forms of biological sequence evolution/mutation. Given a data sequence S, a query P, and a fixed parameter k, the problem is to identify and return the starting locations of all substrings in S which match P with at most k mismatches. The size of each substring returned is $|P|$; no insertion or deletion is allowed in S or P. Further, k is usually a small integer, independent of the sequence and query lengths [Gusfield, 1997, p.200].

One possible solution to this problem is to generate the set *P_all* of all possible strings that can be derived from P by changing up to k symbols in P. This is followed by an exact match search for each string in *P_all*. While this works well when k, the query size, and the alphabet size are relatively small [Gusfield, 1997], in order to provide an efficient and scalable solution, we follow the "the partition approach", proposed in [Wu and Manber, 1992] and further investigated in [Navarro, 1998], [Navarro, 2001].

Figure 14 shows our disk-based, Suffix Tree K-Mismatch search algorithm (STKM), proposed in [Halachev *et al.*, 2009].

```
Algorithm STKM (Sequence S, Index HST, Query P, value k)
  Phase A: Preprocess the query P
  1.  Divide P into k + 1 disjoint substrings, called seeds;
  Phase B: Perform EMS for each seed x_i, where x_i = P[u, v]
  2.  for each seed x_i, where i = 1 to k+1;
  3.    X_i = STEM (S, HST, x_i);
        // X_i has the starting locations of all exact matches of x_i in S
  4.    if (X_i is not empty)
  5.      Phase C();
  6.    end if
  7.  end for
  Phase C: For each L in X_i , extend x_i to check for a k-mismatch
  8.  L = first location in X_i;
  9.  while (not all locations in X_i are examined)
  10.   Align P and S such that P[u] (i.e., x_i[0]) is at S[L];
  11.   set k' = 0; // actual number of mismatches observed so far
  12.   Extend x_i to the left:
            Compare P[u-1, 0] with S[L-1, L-u]
            for each mismatch detected, increment k' by 1 ;
  13.   if (k' ≤ k)
  14.     Extend x_i to the right:
              Compare P[v+1, |P|-1] with S[L+|x_i|, L-u+|P|-1]
                for each mismatch detected, increment k' by 1 ;
  15.     if (k' ≤ k)
  16.       return (k' - mismatch occurs at S[L - u])
  17.   end extending and checking for k-mismatch using current L
  18.   L = next location from X_i;
  19. end while
End STKM
```

Figure 14. STKM Search Algorithm

In the first phase query P is split into $k + 1$ disjoint substrings, which we call *seeds*, denoted by x_i, $i = 1$ to $k + 1$. Since the number of mismatches allowed is at most k, if a k-mismatch between P and a substring K of S exists, then there exists at least one seed in P for which an exact match with a substring of K is found. Thus, the adopted technique for query splitting/seed extraction guarantees 100% k-mismatch recall.

In the next phase, we use STEM algorithm and HST index to compute for each seed x_i, the set X_i containing the starting locations of all exact matches between x_i and substrings in S.

In the last phase, each seed x_i is aligned with S at all locations in X_i and the algorithm recursively extends x_i to the left and to the right, by comparing the unverified characters in P against the corresponding characters in S. The extension process for seed x_i terminates either if all the nucleotides of P are verified, in which case a k-mismatch is found, or if the number of errors observed in the extension phase exceeds k. This k-mismatch search algorithm also provides a 100% k-mismatch precision.

To illustrate the how the STKM algorithm works, consider our running example sequence S = ATGATATGTGAAATAGTAGA$, its HST index, and query P = TGGA. Suppose k = 1. In Phase A, P is arbitrarily divided into two parts, which yields $seed_1$ = TG and $seed_2$ = GA. For $seed_1$, an exact match search is conducted in Phase B, for which we find matches at positions S[1], S[8], and S[6] (see Figure 5), i.e., X_i = 1, 8, 6. In Phase C, this $seed_1$, which corresponds to P[0,1] is first aligned with S[1,2], u = 0, and k' is set to 0, where the variable k' is used to record the number of mismatches detected in the extension process. Note that no left extension for $seed_1$ is possible. Extending P to the right, we find a mismatch, since P[2] = 'G', while S[3] = 'A'. Hence, k' is incremented by 1. Further extending P to the right leads to another mismatch, since P[3] = 'A', while S[4] = 'T'. Hence, k' is incremented by 1, and has value 2, larger than k, the allowed number of mismatches. Thus, no k-mismatch of P starts at S[1].

Next, the algorithm aligns $seed_1$ and S using X_i = 8, i.e., P[0,1] is aligned with S[8,9], u = 0, and k' is set to 0. Extending P to the right, we find a mismatch, since P[2] = 'G', while S[10] = 'A'. Hence, k' is incremented by 1. Further extending P to the right, we find a match between P[3] = 'A' and S[11] = 'A'. The rightmost character in P is processed and k' = 1, which is not greater than k. Thus, we find a k-mismatch of P that

starts at $S[8]$.

The last extension for $seed_1$ aligns $P[0,1]$ with $S[6,7]$. Extending P to the right, we find two mismatches; thus, no k-mismatch of P starts at $S[6]$.

For $seed_2$, an exact match search in Phase B finds matches at positions $S[2]$, $S[9]$, and $S[18]$, i.e., X_i = 2, 9, 18. In Phase C, this $seed_2$, which corresponds to $P[2,3]$ is first aligned with $S[2,3]$, $u = 2$, and $k' = 0$. Note that no right extension for $seed_2$ is possible. Extending P to the left, we find two mismatches ($P[1]$ = 'G' does not match $S[1]$ = 'T'; $P[0]$ = 'T' does not match $S[0]$ = 'A'). Thus, no k-mismatch of P starts at $S[0]$.

Next, the algorithm aligns $P[2,3]$ with $S[9,10]$, $u = 2$, and $k' = 0$. Extending P to the left, we find two mismatches ($P[1]$ = 'G' does not match $S[8]$ = 'T'; $P[0]$ = 'T' does not match $S[7]$ = 'G'). Thus, no k-mismatch of P starts at $S[7]$.

The last extension for $seed_2$ aligns $P[2,3]$ with $S[18,19]$. Extending P to the left, we find a mismatch, since $P[1]$ = 'G', while $S[17]$ = 'A'. Hence, k' is incremented by 1. Further extending P to the left, we find a match between $P[0]$ = 'T' and $S[16]$ = 'T'. The leftmost character in P is processed and $k' = 1$, which is not greater than k. Thus, we find a k-mismatch of P that starts at $S[16]$.

At this stage, all seeds are processed and the search is completed. In total, we found two 1-mismatch occurrences of P in S, starting at locations $S[8]$ and $S[16]$.

Note that in [Navarro, 1998], the authors also study the problem of how to split P in such a way that the number of extensions done in the last phase is reduced and thus the overall search time is improved. We partition P to $k + 1$ seeds of approximately the same length, i.e., we do not optimize their lengths as suggested in [Navarro, 1998]. Thus, the observed HST k-mismatch search times are more representative of HST index structure

quality, rather than descriptive of the k-mismatch algorithm itself. Note that our goal in this work is to evaluate the suitability of the HST index to support various search tasks; further improving the efficiency of the k-mismatch search algorithm by using advanced query splitting techniques is something we plan to investigate as part of our future work.

We now study the disk I/O performance of STKM, using the same analysis basis as for STEM. For conducting the k-mismatch search in Phase A, STKM does not access the sequence or its index. In Phase B it requires access to both, and in Phase C it accesses only the sequence.

Similar to STEM, the STKM algorithm starts by reading the entire sequence (of size n bytes, assuming 1 byte per character) from the disk, which requires $\lceil n/B \rceil$ disk accesses, where B is the page size. In Phase C, no additional disk accesses are required, since the sequence S remains in the memory after the initial reading in Phase B.

Given a query P of size m bytes, in Phase B STKM performs the two-step STEM search for each of the $k + 1$ disjoint seeds, x_i, where $|x_i| \approx m/(k + 1)$. In the first step (i.e., Phase 2 of STEM), for each of the $k + 1$ query seeds, a downward ST traversal is performed. As discussed in Section 5.1., the number of disk accesses for this is at most equal to the number of characters in the seed. Since the sum of the sizes of all seeds is exactly m, the total number of disk accesses for all seeds is at most m in the first step of Phase_B. To analyze the number of disk accesses in the second step (i.e., Phase 3 of STEM), let z_i denote the number of exact matches of seed x_i in S. Hence, the number of sequential disk accesses performed for each seed x_i in the second step is at most $(16*z_i)/B$, as discussed in Section 5.1. There are $k + 1$ seeds, and therefore in the second step of Phase B, the total number of disk accesses for all seeds is at most $(k + 1)(16*z)/B$,

where z is the sum of all z_is. Usually k, the number of allowed errors, is much smaller than m, the number of characters in the query, but the worst case is $k = m - 1$, when STKM must return the starting positions of all substrings in S, which match at least 1 character from P. Thus, in the worst case, there are $((m\text{-}1)+1)(16*z)/B = (16*m*z)/B$ disk accesses. In total, there are at most $m + (16*m*z)/B$ disk accesses per query P in Phase B of STKM. Thus, the k-mismatch search algorithm requires at most $O(n/B + m + mz/B)$ disk accesses for query P.

The time complexity of STKM is determined as follows. Phase A is done in constant time. Phase B, i.e., finding the exact matches for all seeds, requires $O(m + z)$ time, where z is the total number of seeds of P found in S. In Phase C, we extend each seed to the left and right and compare the unverified characters from P to the corresponding characters in S. Each seed's occurrence requires at most $(m - m/(k+1))$ character comparisons. Hence, in Phase C, we perform at most $O(z*(m - m/(k+1))) = O(zm)$ constant time operations. In total, STKM requires $O(m + z + zm)$ time. In the worst case, a seed occurrence can be found starting at each location in S, i.e., $z = n$. Thus, the time complexity of the STKM algorithm is $O(mn)$ in the worst case.

5.3 Structured Motif Search Algorithm (EMOS)

In this section, we study another application of our HST index. A fundamental task in bioinformatics is searching in new sequences for previously known information, expressed as *structured motifs*. Examples of potential applications include searching for composite regulatory binding sites in DNA sequences and finding *long terminal repeat* (*LTR*) *retrotransposons*, which have significant presence in typical mammalian genome

and are believed to have major impact on genome structures and functions [Feschotte *et al.*, 2002; McCarthy and McDonald, 2003].

A structured motif consists of several simple motifs, interleaved by variable-length bounded gaps. Each simple motif can be represented either as a string of symbols from a specific alphabet (*pattern* representation), or as a matrix which gives the probability of observing a specific nucleotide at each position in the simple motif (*profile* representation). A gap is represented as $[x, y]$, which denotes the minimum and the maximum gap sizes allowed between two adjacent simple motifs. This structured motif model provides a suitable way for simultaneously searching for several (interrelated) DNA sequences, while accounting for some possible evolutionary changes.

Consider the following sample structured motif $SM = M_1[2,5]M_2[6,7]M_3$, taken from [Jurka *et al.*, 2005]. In Figure 15, rows 2 to 5 represent each simple motif as a profile, while row 6 gives their corresponding pattern representations using the IUPAC alphabet. Following [Policriti *et al.*, 2004; Zhang and Zaki, 2006], we adopt the IUPAC alphabet for pattern representation of the structured motif, and use the DNA alphabet {A,C,G,T} for sequence data to be searched. The mapping convention between the DNA bases and the corresponding IUPAC symbols is shown in Figure 16. To avoid confusion, we refer to characters in a pattern motif as *symbols* and to sequence characters as DNA bases, or *bases* for short.

Bases	M_1									[2,5]	M_2		[6,7]	M_3		
A	2	12	17	1	11	1	35	0	24		1	0		3	1	35
C	0	10	8	5	2	0	0	19	0		0	25		5	35	1
G	2	5	5	2	10	34	1	0	0		26	11		0	0	0
T	32	9	6	28	13	1	0	17	12		9	0		28	0	0
IUPAC	D	N	N	N	N	D	R	Y	W	[2,5]	D	S	[6,7]	H	M	M

Figure 15. Sample Structured Motif

Bases	A	C	G	T	U	A,G	C,T	G,T	A,C	G,C	A,T	C,G,T	A,G,T	A,C,T	A,C,G	A,C,G,T
Symbol	A	C	G	T	U	R	Y	K	M	S	W	B	D	H	V	N

Figure 16. The IUPAC Alphabet [NC-IUB, 1992]

Given a sequence S and a structured motif SM, the goal of the search is to find the starting positions pos in S at which a match between the query SM and a substring of S occurs. For example, consider our sample query SM and a sample substring $S[pos, pos+25]$ of S starting at position pos, shown in the first row in Figure 17. The next four rows in the figure show 4 matches between SM and $S[pos]$, for the gap sizes {3,6}, {3,7}, {5,6}, and {5,7}, respectively.

$S[p, p+25]=$	T	A	C	G	T	A	A	T	T	G	G	A	A	C	A	C	G	C	A	T	A	C	A	A	A	A
$SM=$	D	N	N	N	N	D	R	Y	W	-	-	-	D	S	-	-	-	-	-	-	H	M	M			
$SM=$	D	N	N	N	N	D	R	Y	W	-	-	-	D	S	-	-	-	-	-	-	-	H	M	M		
$SM=$	D	N	N	N	N	D	R	Y	W	-	-	-	-	-	D	S	-	-	-	-	-	-	H	M	M	
$SM=$	D	N	N	N	N	D	R	Y	W	-	-	-	-	-	D	S	-	-	-	-	-	-	-	H	M	M

Figure 17. Matching a *SM* with a substring of *S*

Formally, a structured motif SM consisting of two or more *simple motifs* M_k separated by gaps of possibly variable lengths, is represented as $SM = M_1[i_1, j_1]M_2[i_2, j_2]M_3..M_{n-1}[i_{n-1}, j_{n-1}]M_n$, where $i_k \leq j_k$. The gap values i_k and j_k indicate respectively the minimum and maximum gap sizes allowed between the last symbol in M_k and the first symbol in M_{k+1}.

In the related literature, we can find several variants of the structured motif search, depending on the definition of match and the constraints imposed by the gaps. We next review these variants and define the problem addressed by our solution.

An *exact match* of SM in S is defined as an "exact" match between all the symbols in SM and the corresponding bases in a substring of S, satisfying the gap constraints. Note that when using the IUPAC representation for a motif and using DNA representation for the sequence data, the notion of "exact" match between a motif symbol and a sequence base is defined by the mapping scheme shown in Figure 16. Examples of exact matches

of *SM* in *S* are shown in Figure 17. Even though this "exact" matching allows some flexibility and is approximate in nature, some applications may require even more relaxed matching. One such example is allowing for errors (i.e., insertions, deletions, and substitutions of symbols/bases) when comparing simple motif symbols and their corresponding DNA bases. This variation of the structured motif search is referred to as *approximate match* [Zhang and Zaki, 2006]. For example, in our sample *SM* and *S* (Figure 17), suppose the number of errors allowed per simple motif is at most 1. Then approximate match search returns a match for gap sizes {2, 6}, in addition to the results returned by exact match. Another type of approximate search allows up to q' (out of all n) simple motifs to be missing. This variant of the problem is referred to as *q-occurrence search* [Policriti *et al.*, 2004; Zhang and Zaki, 2006] and its goal is to find all occurrences of *SM* in *S*, where at least $q = n - q'$ simple motifs are matched. For our sample *SM* and *S*, this type of search for $q = 2$ will return a match for gap sizes {4, 6}, in addition to the results returned by exact match. In this work, we are interested in *exact match* motif search, for which all IUPAC symbols in *SM* must match (according to the mapping scheme in Figure 16) the corresponding DNA bases in *S*, and no missing simple motifs are allowed.

Considering the gap constraints, the *fixed* motif search problem is the simplest case in which $i_k = j_k$, for all k in [1, n-1], and every i_k is known in advance and is positive. When at least one i_k is different from j_k, the problem is referred to as *structured* motif search. Yet another version of the problem allows negative values for i_k, with the restriction that the absolute value of i_k is smaller than the size of M_k. This amounts to allowing partial overlap between M_{k+1} and some of the rightmost symbols of M_k, and hence adds more

flexibility to the search. This variant is referred to as *overlapping structured* motif search and is addressed in this work.

To summarize, the problem for which we propose a solution using our HST index is the exact match overlapping structured motif search, where structured motifs are represented as patterns over the IUPAC alphabet, and no missing simple motifs are allowed.

Our EMOS (Exact Match, Overlapped Structured motif search) algorithm [Halachev and Shiri, 2008], takes as input a structured motif *SM* represented as an IUPAC pattern, a DNA sequence *S*, and its HST index and outputs all starting positions in *S* at which we find an exact match between the *SM* and a substring of *S*, together with the length of the match (Figure 18).

Algorithm EMOS(Sequence *S*, Index *HST*, Structured Motif *SM*)
0. Read Sequence S from disk to memory
Step 1: Preprocess the Structured Motif *SM*
1. Select the anchor simple motif (M_a)
Step 2: Find all occurrences of M_a in *S*
2. *ARS* = *STTD64 root*; *//ARS is Answer Root Set*
3. for each symbol *i* in M_a (starting from M_a[0]){
4. *dna_set* = convert M_a[*i*] to its corresponding DNA base(s); *//see Figure 16*
5. for each STTD64 node (*node_old*) in *ARS*
6. for each *node_old*'s outgoing edge
7. if (its label match one base in *dna_set*)
8. then insert in *ARS* this edge's destination node (*node_new*);
9. delete *node_old* from *ARS*;
10. } //at this point *ARS* contains all answer roots for M_a
11. *occ_*M_a = NULL; *//occ_*M_a *contains start locations of all exact matches of* M_a *in S*
12. for each answer root (*AR*) node in *ARS*{
13. for each leaf in subtree rooted at *AR*
14. compute *S* location, add it to *occ_*M_a;
15. }
Step 3: For each *L* in *occ_*M_a, align *SM* with *S*, check the other simple motifs
16. *L* = first location in *occ_*M_a;
17. while (not all locations in *occ_*M_a are examined){
18. *sm_len* = $|M_a|$;
//Explore SM to right and left of M_a, *matching remaining motifs*
19. extendRight(*a*, *L*);
20. }**end EMOS**

```
extendRight(v, loc){
 //M_v - verified simple motif; loc - a starting location of M_v in S;
 right = v + 1;
 while (right <= n){ //n is the number of the rightmost simple motif in SM
  for each allowed gap value x, i_v <= x <= j_v (starting from i_v){
   align M_right and S such that M_right[0] is at S[loc + |M_v|+x];
   compare M_right symbols with the corresponding S bases
   if (mismatch) then x++; //consider next gap value
   else //exact match for M_right found
    sm_len = sm_len + |M_right| + x;
    loc = loc+|M_v|+x; //loc points to the leftmost M_right symbol
    if (right < n) //not reached the rightmost simple motif yet
     extendRight(right, loc);
    else //a match for rightmost simple motif found
     extendLeft(a, L);
  }//end for
 }//end while
}end extendRight


extendLeft(v, loc){
 left = v - 1;
 while (left >= 1){
  for each allowed gap value y, i_left <= y <= j_left (starting from i_left){
   align M_left and S such that M_left[0] is at S[loc - |M_left| - y];
   compare the non-overlapped M_left symbols with the corresponding S bases
   if (mismatch) then y++; //consider next gap value
   else //exact match for M_left found
    sm_len = sm_len + |M_left| + y;
    loc = loc - |M_left| - y; //loc points to the leftmost M_left symbol
    if (left > 1) //not reached the leftmost simple motif yet
     extendLeft(left, loc);
    else //a match for leftmost simple motif found
     return (SM occurs at position S[loc], length = sm_len bases);
  }//end for
 }//end while
}end extendLeft
```

Figure 18. EMOS Search Algorithm

EMOS takes a three-step approach in solving the structured motif search problem. First, based on a heuristic using the information content of each simple motif in *SM*, the algorithm preprocesses the structured motif and selects a simple motif, M_a, estimated to have the fewest number of matches in *S*. In the second step, using the HST index for the sequence *S*, the algorithm retrieves all starting positions in *S* at which a match with M_a

occurs. We refer to these positions as *anchors*. In the final step, the structured motif is aligned with the sequence, using the exact occurrences of M_a as anchors, and the symbols of the remaining motifs are compared with the corresponding sequence bases. A match of *SM* in *S* occurs if, for a unique and allowed combination of the gap values, we find a match for all simple motifs of *SM* in *S*.

To illustrate the how the EMOS algorithm works, consider the following problem of finding all exact match occurrences of *SM* = WN[-1,2]KW[2,4]Y in our sample sequence *S* = ATGATATGTGAAATAGTAGA$. For clarity of the exposition, we will use the graphical ST representation of the index (Figure 5).

In Step 1 of EMOS, we preprocess *SM* to select a suitable simple motif, as the anchor motif M_a. Intuitively, a suitable motif is a simple motif with smallest number of exact matches in *S*, which will be used as anchors in the subsequent phase. Since we do not know in advance the number of exact matches of each simple motif in *S*, we employ the following heuristic for selecting M_a. By reading *SM* once, we compute the *selectivity power* (*SP*) of each simple motif, by considering the information content of its symbols, according to Figure 16. For example, the selectivity powers of the three simple motifs of the *SM* are computed as follows: $SP(M_1) = SP(\text{WN}) = {}^2/_4 * {}^4/_4 = 0.50$; $SP(M_2) = SP(\text{KW}) = {}^2/_4 * {}^2/_4 = 0.25$; $SP(M_3) = SP(\text{Y}) = {}^2/_4 = 0.50$. Assuming uniform distribution of the DNA bases in *S*, the expected number of exact matches is estimated to be around 50% of the size of *S* for M_1, 25% for M_2, and 50% for M_3. Thus, we select M_2 as M_a. A drawback of this selection method is that in practice the distribution of bases in nucleotide sequences is not strictly uniform. However, our experimental results indicate that although not optimal, the employed heuristic for selecting M_a provides 2 to 3 times overall search

speedup, compared to picking M_a randomly, as discussed in more detail in the experimental section. Note that since the *SP* for each simple motif is strictly greater than 0, this step cannot lead to a premature and incorrect conclusion that *SM* does not occur in *S*, i.e., the adopted heuristic does not contravene the 100% recall and precision of our EMOS algorithm.

In Step 2, we find all exact match occurrences of the selected M_a in *S* in two phases. In the first phase (lines 2-9), starting from the ST *root* and the first symbol of M_a, the STTD64 is traversed downwards, matching the corresponding bases for each of the M_a symbols. At the end of the traversal, the set *ARS* contains the *answer roots* for *all* DNA queries that can be derived from the IUPAC representation of M_a and for which at least one occurrence in *S* is found. In the second phase (lines 11-14), each answer subtree is traversed and from each leaf node, we obtain a starting location of M_a in *S*. Consider our selected M_a = KW, where K = T or G, and W = A or T (see Figure 16). Initially *ARS* = {*root*} (Figure 18, line 2). The first iteration of the FOR loop (lines 3-9) converts the first M_a symbol K to the *dna_set* = {T, G}. All outgoing edges from the *root* (Figure 5) are examined for a match between their label and the bases in the *dna_set*. As a result, at the end of this iteration we find that *ARS* = {node *2*, node *3*}. For the second M_a symbol, *dna_set* = {A, T} and by considering the outgoing edges of each of the *ARS* nodes, the set *ARS* is updated to {node *20*, node *29*, node *30*}. Since all M_a symbols are processed, the second phase starts.

The answer subtree rooted in node *20* has three leaf nodes – nodes *25*, *27*, and *28*, representing starting locations *S*[4], *S*[13], and *S*[16], which are added to set occ_M_a. The answer subtree rooted in node *29* contains leaf nodes *31*, *32*, and *33*, representing starting

locations $S[2]$, $S[9]$, and $S[18]$, which are also added to occ_M_a. Last, the answer subtree rooted in node *30* contains leaf nodes *34* and *35*, representing starting locations $S[7]$ and $S[15]$. So, at the end of the second phase, the starting locations of all exact matches of M_a in S are collected in the set occ_M_a = {4, 13, 16, 2, 9, 18, 7, 15}.

In Step 3 of EMOS, we start with the first verified simple motif (M_a) as an anchor and recursively explore the adjacent simple motifs on the right (Figure 18, function *extendRight*). If for a particular combination of gap values, exact matches for all simple motifs to the right of M_a are found, the algorithm explores recursively the left adjacent simple motifs of M_a in a similar manner (i.e., function *extendLeft*). If for a particular combination of gap values, exact matches for all simple motifs to the left of M_a are found, an exact match for the whole *SM* is found, and its starting position in S and its length are returned. Considering our ongoing example, the first location of occ_M_a set is $L = 4$. In Step 3, the algorithm sets *sm_len* = $|M_a| = 2$, and calls the function *extendRight* with $a = 2$ and $L = 4$ (line 18). For $L = 4$, EMOS finds three occurrences of *SM* in S: one of size 6 bases starting at position $S[3]$; one of size 8 bases starting at position $S[1]$; an one of size 9 bases starting at position $S[0]$, as shown in Figure 19. Overall, there are 14 exact match occurrences of *SM* in S, as shown in Figure 20.

extendRight(2, 4):
right := 2+1 = 3,
x = 2 => compare $M_3[0]$ = 'Y' with S[4+2+2] = S[8] = 'T' → match
all M_3 symbols matched, sm_len = 2+1+2 = 5, M_3 is rightmost simple motif
extendLeft(2, 4):
left := 2-1 = 1,
y = -1 => compare $M_1[0]$ = 'W' with S[4-2-(-1)] = S[3] = 'A' → match
all non-overlapped M_1 symbols matched,
sm_len = 5+2 +(-1) = 6, loc = 4-2-(-1) = 3, M_1 is leftmost simple motif
print: **SM occurs at position S[3], length = 6 bases**
y = 0 => compare $M_1[0]$ = 'W' with S[4-2-0] = S[2] = 'G' → no match;
y = 1 => compare $M_1[0]$ = 'W' with S[4-2-1] = S[1] = 'T' → match
compare $M_1[1]$ = 'N' with S[2] = 'G' → match
all non-overlapped M_1 symbols matched,
sm_len = 5+2+1 = 8, loc = 4-2-1 = 1, M_1 is leftmost simple motif
print: **SM occurs at position S[1], length = 8 bases**
y = 2 => compare $M_1[0]$ = 'W' with S[4-2-2] = S[0] = 'A' → match
compare $M_1[1]$ = 'N' with S[1] = 'T' → match
all non-overlapped M_1 symbols matched,
sm_len = 5+2+2 = 9, loc = 4-2-2 = 0, M_1 is leftmost simple motif
print: **SM occurs at position S[0], length = 9 bases**
x = 3 => compare $M_3[0]$ = 'Y' with S[4+2+3] = S[9] = 'G' → no match
x = 4 => compare $M_3[0]$ = 'Y' with S[4+2+4] = S[10] = 'A' → no match
//end **extendRight**(2, 4);

Figure 19. Partial Illustration of Step 3 (EMOS)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
S =	**A**	**T**	**G**	**A**	**T**	**A**	**T**	**G**	**T**	**G**	**A**	**A**	**A**	**T**	**A**	**G**	**T**	**A**	**G**	**A**	**$**	*pos*
SM =				W	K	W	-	-	Y													Found at *S*[3]
SM =		W	N	-	K	W	-	-	Y													Found at *S*[1]
SM =	W	N	-	-	K	W	-	-	Y													Found at *S*[0]
SM =		W	K	W	-	-	Y															Found at *S*[1]
SM =	W	N	K	W	-	-	Y															Found at *S*[0]
SM =		W	K	W	-	-	-	-	Y													Found at *S*[1]
SM =	W	N	K	W	-	-	-	-	Y													Found at *S*[0]
SM =									W	K	W	-	-	Y								Found at *S*[8]
SM =							W	N	-	K	W	-	-	Y								Found at *S*[6]
SM =						W	N	-	-	K	W	-	-	Y								Found at *S*[5]
SM =							W	K	W	-	-	-	-	Y								Found at *S*[6]
SM =						W	N	K	W	-	-	-	-	Y								Found at *S*[5]
SM =					W	N	-	K	W	-	-	-	-	Y								Found at *S*[4]
SM =				W	N	-	-	K	W	-	-	-	-	Y								Found at *S*[3]

Figure 20. Exact Match Occurrences of *SM* = WN[-1,2]KW[2,4]Y in *S* = ATGATATGTGAAATAGTAGA$

Searching for structured motifs represented as patterns is an active research area [Mehldau and Myers, 1993; Myers, 1996; Navarro and Raffinot, 2003; Policriti *et al.*, 2004; Zhang and Zaki, 2006]. Next, we briefly review related work and compare their performance with EMOS.

SMaRTFinder [Policriti *et al.*, 2004] uses a ST index and adopts a two-step approach. It first finds all the occurrences of each simple motif, using the *wotd* suffix tree index (Section 3.1) for the sequence data. In the second step, SMaRTFinder solves a constraint satisfaction problem, by building a constraint graph for all possible pairs of simple motifs occurrences (represented as nodes) which locally satisfy the gap constraints. Subsequently this graph is pruned only to *feasible* nodes (i.e., nodes that represent occurrences of simple motifs that certainly belong to a match for the structured motif), and the set of all structured motif matches is obtained by a depth-first traversal of the pruned graph. The experimental results in [Policriti *et al.*, 2004] indicate a significant search time advantage of SMaRTFinder over Anrep [Mehldau and Myers, 1993; Myers, 1996], when searching for a randomly generated set of 1,000 structured motifs in a 5 MB DNA sequence. Further, the SMaRTFinder exhibits linear search time with respect to the number of matches found, while the performance of Anrep depends strongly on the success of its statistical optimization of the backtracking match algorithm.

In a recent work [Zhang and Zaki, 2006], the authors proposed the SMOTIF technique for structured pattern and profile motif search. It consists of several algorithms, which support exact match, approximate match, and q-occurrence motif search operations. There are two alternative implementations for structured pattern search:

SMOTIF1 and SMOTIF2. Below, we review their approach to the exact match motif search problem.

As a first step, SMOTIF1 scans the sequence to be searched and converts it into an equivalent inverted format [Zaki, 2000; Zaki, 2001], where each character in the sequence is associated with its *post-list* – a sorted list of the positions at which the base occurs in the sequence. Also, the structured motif is converted into its SMOTIF representation, by adding a gap [0,0] between adjacent symbols within each simple motif, and then consolidating the gaps, if possible. For example, the motif GCN[0,1]TB is converted into G[0,0]C[1,2]T[0,0]B. The second step in SMOTIF1 starts from the last two motif symbols (T and B in this example) and computes the post-list of T[0,0]B, using *positional joins* over the post-lists of T and B (each of which viewed as a union of the post-lists of their corresponding matching bases, computed in the first step). The result essentially is the list of all starting positions of T[0,0]B in the sequence. Next, the algorithm recursively expands the positional join process, by considering the first unprocessed symbol to the left (in our example, C), and performs a positional join over its post-list and the post-list of T[0,0]B. Upon completion of the recursive positional join process, the post-list of the entire structured motif is obtained, i.e., the list of all starting positions of the structured motif in the sequence. If required by the application, the set of matching positions for each symbol in the motif can be recovered.

In contrast to SMOTIF1, which performs positional joins on the post-lists of individual motif symbols, SMOTIF2 performs the positional join process on the post-lists of the whole simple motifs. That is, the post-list (i.e., the starting positions) of each simple motif is obtained by *lazy* construction of the *wotd* ST index (i.e., the same index

as in SMaRTFinder). While this first step is the same for SMOTIF2 and SMaRTFinder, the main difference between them is in the second step. The positional join technique of SMOTIF2 proves to be more efficient than the constraint satisfaction technique used in SMaRTFinder.

The experimental results show that SMOTIF1 and SMOTIF2 are respectively up to 18 and 4 times faster than SMaRTFinder searching for three real-life motifs in chromosome 1 of *A. Thaliana* [Zhang and Zaki, 2006]. Also, a more comprehensive comparison of performance of the three search techniques is done, by searching chromosome 20 of *Homo sapiens* for a set of 100 random structured motifs. Again, SMOTIF1 and SMOTIF2 are 6 and 4 times faster than SMaRTFinder, respectively. It is also shown that SMOTIF1 performs significantly better than SMOTIF2 when no missing simple motifs are allowed, i.e., the search problem we are addressing with EMOS. Further, a problem in SMOTIF2 (as well as in SMaRTFinder) is that in the first step, it searches for the exact match occurrences of all simple motifs of a *SM*. In case of long sequences and/or several simple motifs that have low information content, this may lead to enormous intermediate output which may not fit in the main memory. As a result, SMOTIF2 and SMaRTFinder run out of memory in such cases and cannot conclude the *SM* search. Thus, we regard SMOTIF1 as the best known solution for exact match overlapping structured motif search, both in terms of efficiency and scalability. We use it as a benchmark with which we compare the performance of our EMOS algorithm, studied in the next chapter.

6 Experiments and Results

Searching in DNA and protein sequence data is at the core of bioinformatics applications. Recognizing this need, our aim is to provide an extendable framework for solving various search problems. To this end, we develop a powerful tool to support efficient, scalable, and versatile search in biological sequences on regular desktop computers. Our technique includes a novel suffix tree based index, HST, its construction algorithm, and an extendable set of search applications using the index.

In this section, we evaluate HST and compare it to the best known alternatives, considering index construction time, storage space, search efficiency, scalability, and versatility of the index support.

A popular solution that addresses almost all of the above criteria is Vmatch [Vmatch, 2007], a commercial software tool that implements the enhanced suffix array (ESA) indexing technique [Abouelhoda *et al.*, 2004], described in Section 2.3.2. Due to its fast index construction and efficient and versatile search performance, we regard Vmatch as one of the best solutions for sequences up to 250MB, which is the limit for Vmatch on regular desktops. The executable of Vmatch was made available to us.

The SeqAn library [SeqAn, 2008] provides a general, easy to use, and extendable library of efficient algorithms and data structures for biological sequence analysis. It includes an independent implementation of ESA [Abouelhoda *et al.*, 2004], which supports both memory- and disk-based index construction and search, thus handling sequences up to 4GB on typical desktops. We obtained the svn trunk of SeqAn (revision 3085) implemented in C++ from [SeqAn, 2008].

The executable/source code of the suffix array based LOF-SA technique [Sinha *et al.*, 2008] is not available yet (Dec. 2008), thus we consider an indirect comparison.

TDD [Tian *et al.*, 2005] is used as a baseline technique in related literature (e.g., [Phoophakdee and Zaki, 2007], [Sinha *et al.*, 2008]) for being the first technique to build the suffix tree index for the entire human genome (2.7GB) on a typical desktop. We obtained the C++ source code of its disk-based construction and its memory-based exact match search algorithms from its web site [TDD, 2005]. However, note that the focus of [Tian *et al.*, 2005] was on index construction for sequences up to 4GB; at this stage its search performance is not optimized.

Trellis [Phoophakdee and Zaki, 2007] is included in our comparison study so that we can compare the performance of our proposal to another suffix tree based technique that is capable of creating the index for and searching directly in the entire human genome. We obtained the C++ source code of its disk-based construction algorithm from http://www.benjarath.com/ and its disk-based exact match algorithm from one of the authors [Phoophakdee, personal communication]. Note that Trellis supports processing DNA sequences only.

The source code of SMOTIF1, implemented in C++, is available at [SMOTIF, 2007]. We use SMOTIF1 as a benchmark for the structured motif search problem, since it has been shown that it outperforms the existing solutions. Note that none of Vmatch, SeqAn, TDD, and Trellis supports such functionality at this time. Also note that SMOTIF1 does not construct a persistent index, thus only a search time comparison with the HST based solution is provided.

The construction algorithm for our proposed HST index, as well as the search algorithms using the index, are written in C. We have implemented our technique in a software package, called FASST (Fast And Scalable Search Tool), a web-based interface to which is available at [FASST, 2008].

All experiments are conducted on typical 32-bit desktop computer with Intel Pentium 4@3 GHz, 2GB RAM, a single 300GB HDD, and 2MB L2 cache, running Linux kernel 2.6, with page size of 4096 bytes and no swap space. We compiled the source code of all programs (except Vmatch) using *gcc/g++* 4.1.1, with the option –O3 enabled. In our experiments, we allow all algorithms to use as much as possible of the available 2GB RAM. All time measures reported are the elapsed, wall clock times.

Our experimental results lead us to believe that our proposal, FASST, bridges the gap between efficiency (i.e., Vmatch) and scalability (i.e., SeqAn, TDD, and Trellis). The rest of this chapter is organized as follows. First, we introduce the biological sequence data used in our experiments. Next, we present our results on index construction times for FASST, Vmatch, SeqAn, TDD, and Trellis techniques. Third, we report storage requirements of these solutions. Last, we focus on the search performance of our technique, comparing FASST to Vmatch, SeqAn, TDD, Trellis, and SMOTIF1.

6.1 Biological Sequences used in our Experiments

The biological sequence data we considered in our experiments are presented in Table 1. In our experiments, we consider all 24 human chromosomes, i.e., *chr1* to *chr 22*, and *chrX* and *chrY*. We preprocessed the input DNA sequences by removing symbol N (denoting unknown nucleotides), and hence the DNA alphabet size is 4. The resulting sequences sizes are shown in Figure 21. For SwissProt database, we removed header

lines, new line symbols, and blanks. We used two versions of the TrEMBL database, which were preprocessed in the same way as SwissProt. The resulting sequences, called *sprot*, *trembl_old*, and *trembl_new*, respectively, are over an alphabet of 23 symbols. Note that although 3 bits are sufficient to represent any nucleotide and 5 bits are for any amino acid, we use 1 byte to store each character in the considered DNA and protein sequences.

Table 1. Biological Sequences used in our experiments

DNA Sequences	Protein Sequences
chr1, ..., chr22, chrX, and chrY each of the 24 human chromosomes [NCBI, 2007b], sizes from 25 to 238MB (see Fig. 21)	***sprot*** concatenation of all the protein sequences in SwissProt [ExPASy, 2007], size 92MB
chr1&2 the concatenation of *chr1* and *chr2*, size 441MB	***trembl_old*** concatenation of all the protein sequences in an older TrEMBL version [ExPASy, 2006], size 840MB
HG the entire human genome obtained by concatenating *chr1* to *chrY*, size 2.7GB	***trembl_new*** concatenation of all the protein sequences in a newer TrEMBL version [ExPASy, 2007], size 1.3GB

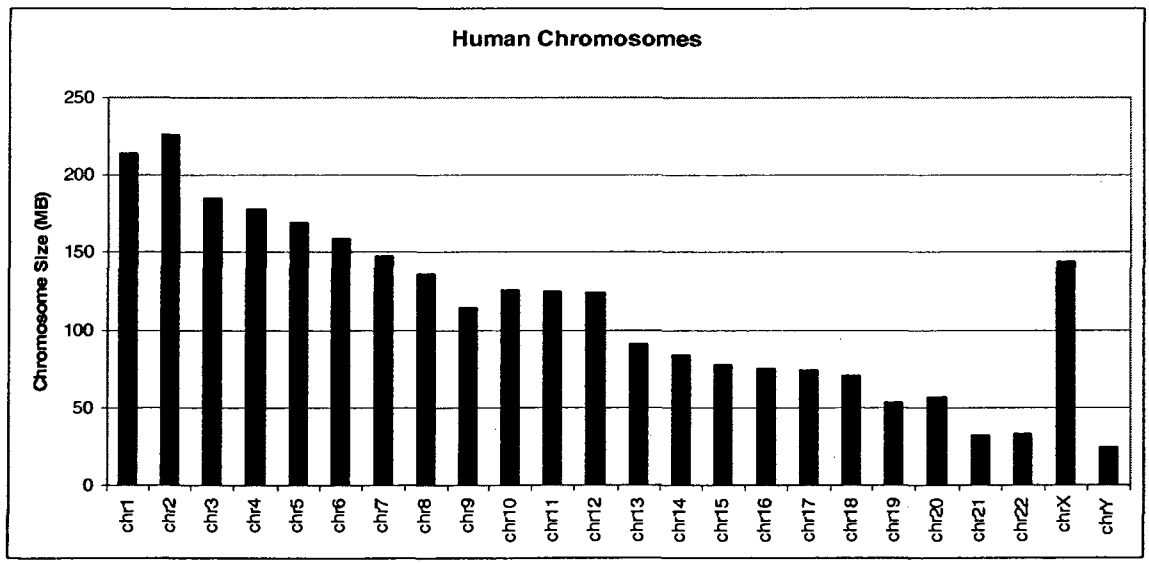


Figure 21. The Size of Human Chromosomes (MB)

Based on the limitations of the various indexing techniques, we group the sequences as follows:

- **Short Sequences** (up to 100MB) - the 11 shortest chromosomes (i.e., *chrY*, *chr22* to *chr13*) and *sprot*;
- **Medium Sequences** (from 100MB to 250MB) - the remaining 13 chromosomes (i.e., *chrX*, *chr12* to *chr1*);
- **Long Sequences** (from 441MB to 1.3GB) - *chr1&2*, *trembl_old*, *trembl_new*;
- **Very Long Sequences** (2.7GB) – the entire human genome (i.e., *HG*).

Next, using these sequences we study and discuss the index constructions times of the various indexing techniques.

6.2 Index Construction Times

The index construction times for the considered techniques are shown in Table 2, where the fastest times are shown in bold.

Table 2. Index Construction Times (in minutes)

Size Range	Sequence	Type	FASST	Trellis	TDD	Vmatch	SeqAn
Short (< 100MB)	11 shortest human chromosomes (total size 674MB)	DNA	45	34	24	**13**	67
	sprot (92MB)	Protein	3	N/A	3	**1**	?
Medium (100-250MB)	13 longest human chromosomes (total size 2,052MB)	DNA	145	121	133	**60**	?
Long (441MB-1.3GB)	*chr1&2* (441MB)	DNA	32	**28**	(90)?	N/A	?
	trembl_old (840MB)	Protein	47	N/A	**41**	N/A	?
	trembl_new (1.3GB)	Protein	107	N/A	?	N/A	?
Very Long (2.7GB)	*HG* (2.7GB)	DNA	16 hrs	**13 hrs**	30 hrs*	N/A	?

N/A stands for "Not Applicable"; ? indicates the task could not be completed; * as reported in [Tian *et al.*, 2005]

The first row reports the total construction time for the 11 shortest chromosomes, of total size 674MB, and the second row reports the total construction time for constructing

the indexes of the remaining 13 chromosomes, of total size 2,052MB. For creating the LT indexes for all DNA data sequences included in our experiments, we used $p = 7$ nucleotides. This value represents a suitable trade-off between large p values, advantageous for reducing the number of disk I/O operations in Phase 2 of STEM, and small p values, leading to increased reuse of STTD64 buffer content. Also, it is the shortest practical query length. An interesting fact we found is that all 16,384 unique DNA strings of size $p = 7$ nucleotides occur at least twice in each human chromosome. For protein sequences, we use $p = 2$, since this is the shortest practical query length.

As can be seen from the table, for the short and medium sequences it can handle, Vmatch is 2 to 3 times faster in constructing all of its index tables compared to the ST based techniques. This is explained by the fact that Vmatch is a memory-based technique, which does not use buffering and hence no buffer overhead is incurred. Vmatch is about 4 times faster than memory-based version of SeqAn, which is the slowest of all. The SeqAn module for disk-based ESA index construction did not work properly at the time we conducted the experiments (Dec. 2008), which limits our evaluation of SeqAn to DNA sequences up to 100 MB.

Comparing performance of the three ST based techniques, we note that for short sequences TDD has the fastest construction time. For fairness of comparison, and because the exact match search algorithm does not require suffix links, in creating the Trellis index we have *not* computed them, and the times reported do not include this time, nor the time needed to convert the sequences into the format native to Trellis. Also note that Trellis can process DNA sequences only. The advantage of TDD over Trellis for short sequences is explained in [Phoophakdee and Zaki, 2007] - for short sequences most

of the data structures of TDD fit in memory, thus the disk I/O overhead is reduced. For such sequences, TDD is about twice faster than FASST, due to the two additional steps in FASST index construction – filling the depth values for the STTD64 leaf nodes and creating the lookup table LT.

For medium size sequences (100-250MB), FASST takes advantage of its dynamic buffering scheme, and exhibits construction times comparable to TDD and Trellis.

For longer sequences (> 250MB), while TDD and FASST show comparable performance for *trembl_old*, the version of TDD that was made available did not perform as expected for the remaining sequences we considered. For example, while constructing the HST index in FASST for sequence *chr1&2* took 32 minutes, we had to terminate the TDD construction process after 90 minutes, during which only a small part of the index was constructed. Also, TDD was not able to handle *trembl_new* and *HG* sequences.

A possible reason for this problem of TDD is the use of the leaf and rightmost bitmap arrays, which have to be fully in memory, and thus occupy a significant amount of the available RAM. To compare TDD and FASST for long sequences, we consider the construction time for the entire human genome which took 30 hours by TDD, as reported in [Tian *et al.*, 2005], while it took 16 hours by FASST [Halachev *et al.*, 2009]. We observed that for short and some long sequences, the TDD index construction on our machine is faster compared to what is reported in [Tian *et al.*, 2005], possibly due to hardware and/or software differences. Even accounting for these differences which favors TDD, our experiments indicate that FASST is faster in index construction for long DNA sequences. This is because FASST uses a dynamic buffer management strategy and does not use the bitmap arrays used in TDD.

For DNA sequences of sizes greater than 250 MB, both FASST and Trellis were able to construct the ST index, with only a slight time advantage for Trellis. While for sequences of sizes 200MB and 400MB the index construction times for Trellis observed on our machine are very similar to those reported in [Phoophakdee and Zaki, 2007], Trellis took almost 13 hours to construct the *HG* index on our computer; the time reported in [Phoophakdee and Zaki, 2007] was 4 hours, which may be due to hardware differences.

Based on our experimental results, we may conclude that for a wide range of sequence sizes, the disk-based FASST exhibits index construction times generally comparable to the best ST construction algorithms, TDD and Trellis. The only exception is for sequences shorter than 100MB, where FASST is about 2 times slower. FASST, TDD, and Trellis are scalable with respect to the sequences size; they can handle sequences up to 4GB. However, this comes at a price: they are 2 to 3 times slower in index construction compared to Vmatch, which is a memory-based indexing technique that can handle sequences of up to 250MB.

A related factor that should be considered in comparing the alternative techniques is the index storage requirements, discussed next.

6.3 Index Storage Requirements

In this section, we compare the index storage requirements of FASST, Trellis (without suffix links), TDD, Vmatch, and SeqAn (only for the 11 shortest chromosomes), shown in Table 3. With respect to index storage space, almost all considered techniques showed similar storage requirements.

Table 3. Index Storage Requirements (bytes per character)

Sequence	Type	Sequence Size	FASST	Trellis	TDD	Vmatch	SeqAn
chr1...chrY	DNA	25–238 MB	13.5	27.6	12.7	12.1	13.1
chr1&2	DNA	441 MB	13.4	26.9	?	N/A	?
HG	DNA	2.7 GB	13.5	28.2	19.3*	N/A	?
sprot	Protein	92 MB	12.5	N/A	11.8	12.3	?
trembl_old	Protein	840 MB	12.3	N/A	12.1	N/A	?
trembl_new	Protein	1.3 GB	12.6	N/A	?	N/A	?

* as reported in [Phoophakdee and Zaki, 2007]

For a DNA sequence of size n characters, Vmatch, SeqAn, TDD, and FASST created indexes of size about 12-13n bytes. The exception was Trellis, which required 28n bytes for its index without the suffix links (50n with links). This is because Trellis records a ST branch node using 28 bytes and a leaf node using 8 bytes, while for example HST requires 8 bytes for any ST node (branch or leaf). Note that for long DNA sequences, FASST exhibits constant behavior, while TDD requires up to 19.3 bytes per sequence symbol, as reported in [Phoophakdee and Zaki, 2007].

While for protein sequences the index requirements of Vmatch does not change, the larger alphabet size leads to larger branching factor, and hence smaller number of branch nodes for the suffix tree based FASST and TDD. This results in reduced storage requirements for indexing proteins, which we found to be 12.5n and 12n, respectively. We may thus conclude that all techniques have comparable index storage requirements of about 12-13 bytes per sequence character, except Trellis, whose index is twice larger.

Although important, we should note that the index cost (i.e., construction time and storage space) is incurred only once and is subsequently amortized over numerous search operations, as discussed next.

6.4 Search Time Performance

In the previous two sections, we compared FASST to Trellis, TDD, Vmatch, and SeqAn in terms of index construction time and storage requirements. While they are important factors in evaluating and comparing indexing techniques, these two criteria are not decisive factors for the following reasons.

First, while the indexes constructed are larger compared to other solutions (e.g., basic suffix array and various compressed index representations), they are more desirable due to their versatility and efficiency for various search tasks. Since all indexes (except Trellis) have similar storage requirements, no preference can be made based on the storage space.

Second, the index construction time of FASST is comparable to TDD and Trellis. These three disk-based suffix tree construction algorithms are faster than SeqAn, but are 2 to 3 times slower than the memory-based Vmatch. However, for biological sequences whose content is usually stable, the index construction time is incurred only once and its cost is subsequently amortized over numerous search operations that use the index. Thus, if any of the considered techniques provides faster search than Vmatch, its longer construction times pays off and will be acceptable.

Next, we explain the experimental setup and report the measured search times for exact match, k-mismatch, and structured motif search, together with their analysis. Note that we performed our search experiments using a "cold start" for all techniques, i.e., the sequence data S, its index, and the query sets are residing on disk.

6.4.1 Exact Match Search (EMS)

In this section, we study the performance of our STEM algorithm which implements the exact match search functionality in FASST and compare it to existing solutions. We start by explaining how the query sets used in our experiments are generated. We next present the search times for the considered techniques organized according to sequence size.

As a naming convention for the query sets used in our experiments, we use *q-x-y*, where x represents the query length (in number of nucleotides/amino acids) and y denotes the number of queries in the query set.

For DNA queries, we consider the following five query lengths $x = 7$, 11, 15, 40, and 100. The first three values are used in BLASTn [BLAST, 2008]. The value $x = 40$ is considered as the minimum match/anchor size used in genome alignment programs, such as MUMmer [Delcher *et al.*, 1999]. It is also illustrative for megaBLAST [BLAST, 2008], where the values range from 16 to 64 nucleotides. The value $x = 100$ is used as done in the experimental evaluations in [Phoophakdee and Zaki, 2007] and [Sinha *et al.*, 2008]. We do not consider query lengths greater than 100 nucleotides, since all the considered techniques exhibit constant performance after this value, explained by the fact that for $x > 100$, very few matches are found and increasing the query length does not lead to a measurable change in the amount of work (CPU and disk I/O) performed to answer a query. To evaluate the effect of the number y of queries in a query set, we consider four query set sizes, i.e., $y = 10^2$, 10^3, 10^4, and 10^5. Thus, we have 20 different DNA query sets, from *q-7-*10^2 to *q-100-*10^5.

We used the same method as in [Abouelhoda *et al.*, 2004] in order to generate the DNA query sets for our experiments. That is, for query set *q-x-y* we sampled the longest human chromosome, *chr2*, at random locations to extract $y/2$ words of length x and then

included their reverses. For instance, for query set *q-7-10*^{2} we sampled *chr2* at 50 random locations resulting in 50 strings of length 7 nucleotides. For the other half of the queries in this query set, we included the reverses of these 50 strings. As discussed in [Abouelhoda *et al.*, 2004], this approach is used in order to simulate cases with no exact matches found for some queries.

For protein queries, we consider the following two query lengths $x = 2$ and 3 amino acids, which are the values accepted by BLASTp [BLAST, 2008]. To evaluate the effect of the number y of queries in a query set, we consider three query set sizes, i.e., $y = 100$, 300, and 500. Thus, we have 3 different protein query sets: *q-2/3-100*, *q-2/3-300*, and *q-2/3-500*. The protein query set *q-2/3-100* is obtained by sampling the *sprot* sequence at 100 random locations to extract 50 words of size 2 and 50 words of size 3 amino acids. The other protein query sets, *q-2/3-300* and *q-2/3-500*, were generated in a similar manner by sampling *sprot* at 300 and 500 locations, respectively.

6.4.1.1 EMS Comparison on Short Sequences (up to 100 MB)

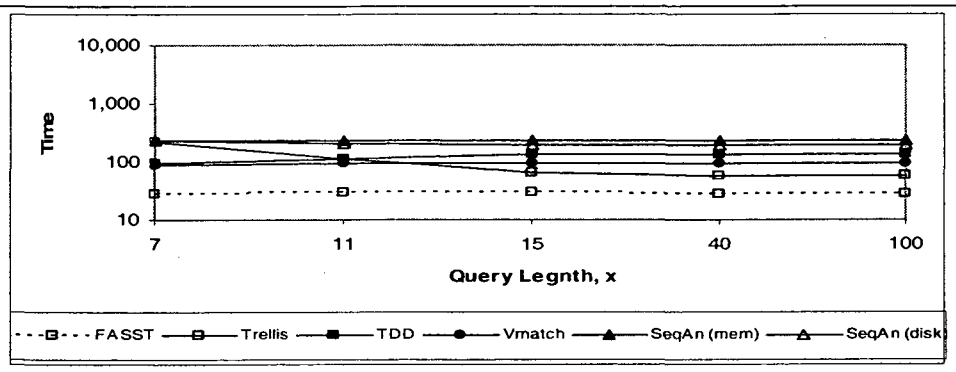
For this class of sequences, all of the considered techniques are applicable, where 100MB is the limit for sequences we can search using TDD and the memory-based version of SeqAn on our machine with 2GB RAM.

In this set of experiments, we used as input the 11 shortest human chromosomes and the 20 DNA query sets described in Section 6.4.1. As discussed in Section 5.1, our HST based exact match search algorithm explicitly sorts the query sets. The reported search times for FASST include this sorting time, which is 0.2 seconds on average, up to 0.5 seconds for the largest query set *q-100-10*^{5}.

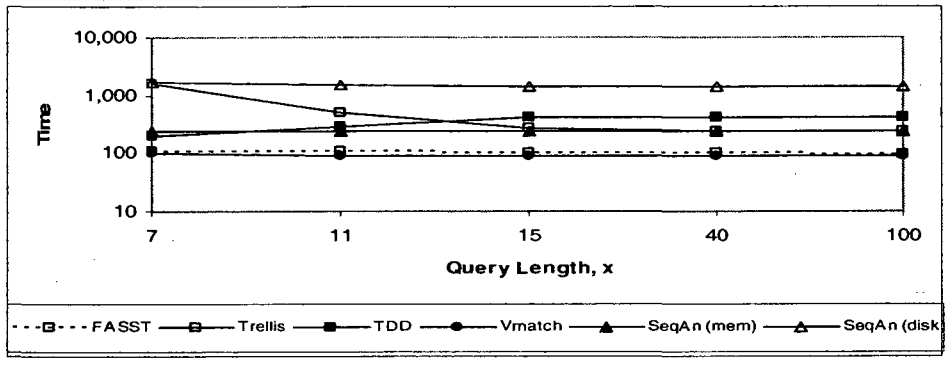
To study the impact of query sets being sorted or not for the other techniques, we performed experiments using unsorted as well as sorted query sets. Our results show that while for short sequences the SA based techniques (Vmatch and SeqAn) are indifferent to the query sets being sorted or not, the ST-based TDD and Trellis benefit much, especially for large number of queries (i.e., *q-x-*10^5), performing respectively about 4 and 8 times faster for sorted sets. The reason is that although Trellis does not use buffering during search and in case of page miss discards the current index page, the OS nevertheless keeps some of the discarded pages in memory, leading to reduced number of "true" disk I/Os. The memory-based TDD search also benefits from the query set being sorted. Even though the improvement gain by TDD and Trellis is OS dependant and cannot be guaranteed, for fairness of comparison we compare the search performance of FASST, which explicitly sorts the query sets, to the performance of all other techniques using the sorted query sets, without adding the sorting time.

The observed search times are depicted in Figure 22 and represent the cumulative search times in all 11 chromosomes for each query set. Based on the results presented in Figure 22, we make the following observations about the performance of the three ST based indexing techniques.

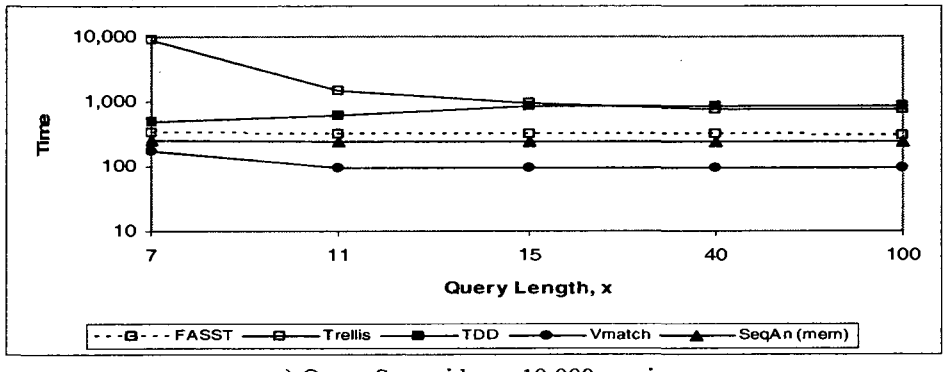
For exact match search in sequences up to 100MB, FASST consistently outperforms both Trellis and TDD, for any query size and any number of queries in the query set, due to its efficient disk I/Os.



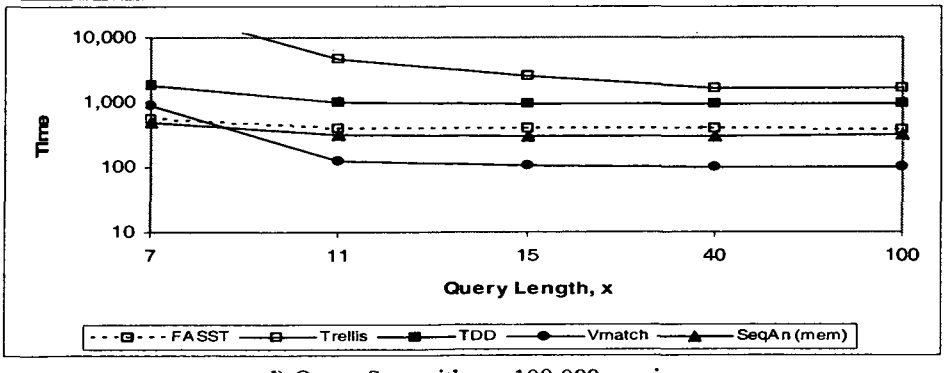
a) Query Sets with y = 100 queries



b) Query Sets with y = 1,000 queries



c) Query Sets with y = 10,000 queries



d) Query Sets with y = 100,000 queries

Figure 22. Exact Match Search Times (in seconds), Short DNA Sequences (<100MB)

In general, Trellis is faster than TDD for long queries, e.g., $x = 40$ and $x = 100$ (also noted in [Phoophakdee and Zaki, 2007] and [Sinha *et al.*, 2008]), except for very large query sets (Figure 22.d), since the I/O cost for reading the entire TDD index into memory is amortized over large number of queries. TDD is also faster than Trellis for short queries, e.g., $x = 7$ and $x = 11$ (also noted in [Sinha *et al.*, 2008]).

Comparing the performance of the two ESA based techniques, we note that Vmatch is about two times faster than the memory-based version of SeqAn. Since we were not able to construct SeqAn indexes for sequences longer than 100MB, we investigate the performance of the disk-based SeqAn search algorithm here as well. For small query sets (Figure 22.a), SeqAn slightly benefits conducting disk-based search. However, it does not scale well with query set size and its performance rapidly deteriorates for larger query sets (e.g., Figure 22.b). For each of the query sets with 10^4 and 10^5 queries, the disk-based SeqAn search takes more than 10,000 seconds (i.e., almost 3 hours), thus its performance is not shown in Figures 22.c) and 22.d).

Overall, for short sequences up to 100MB, the two main competitors are FASST and Vmatch. For query sets with 10^2 queries (Figure 22.a), FASST is about three times faster in search than Vmatch. However, Vmatch scales more smoothly with query set size increase. For answering query sets of size 10^3 (Figure 22.b) the two techniques provide similar performance, while for query sets *q-x-*10^4 and *q-x-*10^5 (Figure 22.c and 22.d) Vmatch is on average three times faster than FASST. One reason for this Vmatch efficiency is that its index is partitioned into a number of tables, and for performing a particular search task, Vmatch uses only the relevant tables. The exact match search

needs only seven of those tables, of total size around $6n$ bytes. On the other hand, FASST reads and keeps in the buffer a substantial part of its $13n$ HST index.

The results for EMS in short protein sequences (i.e., *sprot*) are reported in Table 4. As shown in the table, for protein sequences which Vmatch can handle, FASST is the fastest. One reason for the advantage of FASST over Vmatch when searching in protein sequences is the larger alphabet size of proteins, which results in larger branching factor and hence smaller heights of its HST index.

Table 4. Exact Match Search Times (in seconds), Short Protein Sequences (*sprot*, 92MB)

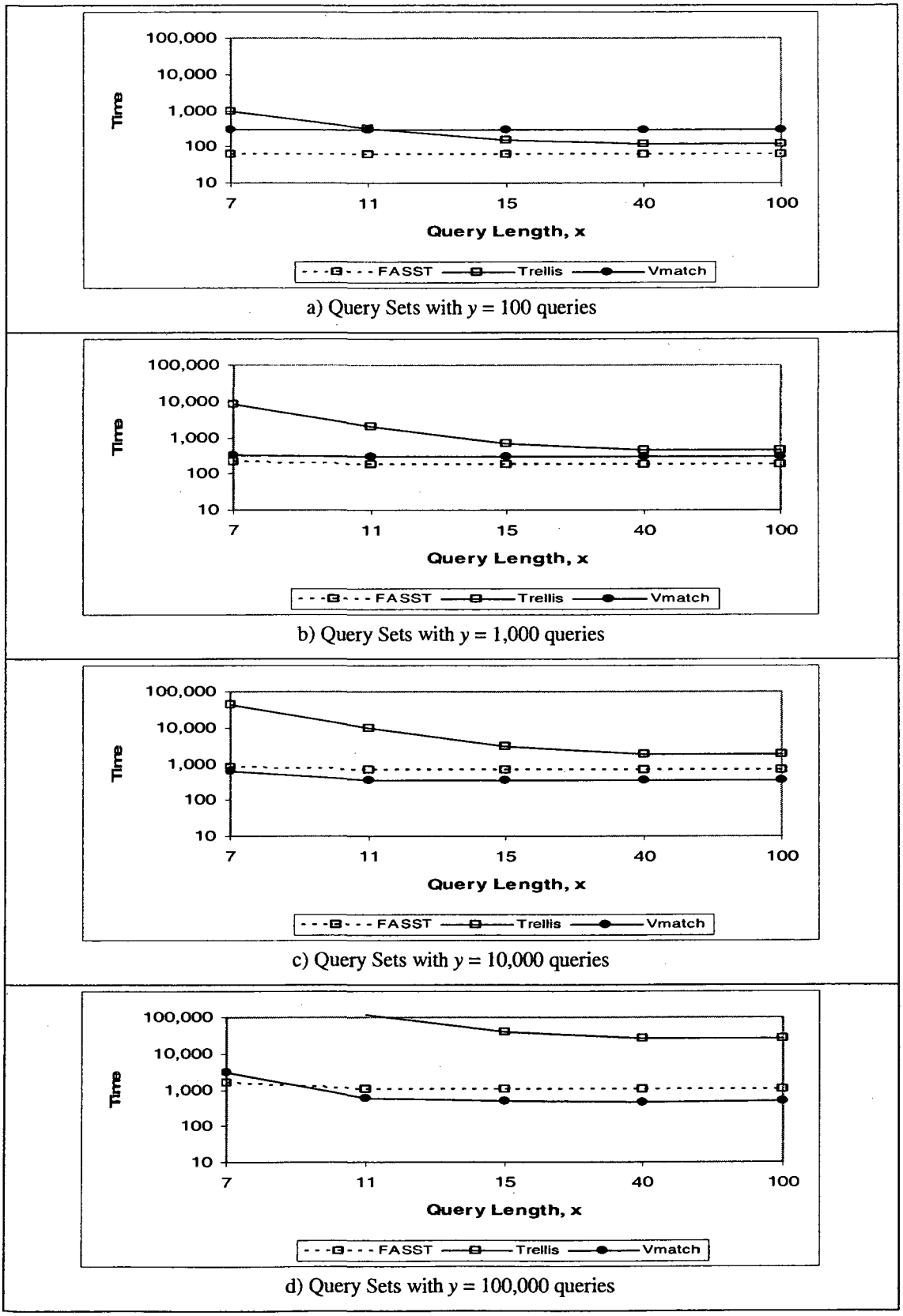
Query Set	FASST	Trellis	TDD	Vmatch	SeqAn
q-2/3-100	**7**	N/A	21	17	?
q-2/3-300	**13**	N/A	37	20	?
q-2/3-500	**18**	N/A	51	24	?

N/A stands for "Not Applicable"; ? indicates the task could not be completed;

Based on the results presented in Figure 22 and Table 4, we can conclude that the EMS performance of FASST for short biological sequences (up to 100MB) is highly competitive to Vmatch, and significantly outperforms Trellis, TDD, and SeqAn. We next study EMS performance of FASST, Trellis, and Vmatch for medium sequences (from 100 to 250MB).

6.4.1.2 EMS Comparison on Medium Size Sequences (100-250MB)

In this set of experiments, we used the 13 longest human chromosomes and the same 20 DNA query sets as for short sequences. For this class of sequences, the applicable techniques are FASST, Trellis, and Vmatch. Figure 23 depicts the cumulative search times in all 13 chromosomes for each query set. As explained earlier, we used sorted query sets for Trellis and Vmatch.



a) Query Sets with y = 100 queries

b) Query Sets with y = 1,000 queries

c) Query Sets with y = 10,000 queries

d) Query Sets with y = 100,000 queries

Figure 23. Exact Match Search Times (in seconds), Medium Size DNA Sequences (100-250MB)

For sanity check, we also ran Trellis and Vmatch with the unsorted query sets and the results reconfirmed our observation made for short sequences - the search is faster using sorted query sets. For example, for medium size sequences and query sets *q-x-10*^{5}, Trellis is up to 4 times and Vmatch is up to 2 times faster using the sorted query sets.

As is the case for short sequences, Trellis performs well only in case of small number of long queries, but even in those cases, it is about two times slower than FASST. Again, the fastest techniques for search in medium size sequences are FASST and Vmatch. Our results also indicate that FASST scales up better with sequence size compared to Vmatch. For small query sets (i.e., *q-x-10*^{2}) and medium size sequences, FASST is about five times faster than Vmatch (Figure 23.a), compared to the three-fold advantage it has for the same query sets and short sequences (Figure 22.a). For query sets of size 10^3, FASST is now about 50% faster than Vmatch. For large query set sizes (i.e., *q-x-10*^{4} and *q-x-10*^{5}), Vmatch is still faster than FASST, but now only twice.

We further evaluate scalability of FASST with respect to sequence size by conducting search experiments in long and very long sequences, presented next.

6.4.1.3 EMS Comparison on Long and Very Sequences (above 250 MB)

In this set of experiments, we compare the performance of FASST, Trellis, and Vmatch for long and very long sequences. For long sequence, we use *chr1&2*, obtained by concatenation of human chromosomes 1 and 2, and for very long sequence, we use *HG*, i.e., the entire human genome, obtained by concatenation of all the 24 human chromosomes.

As already discussed, FASST and Trellis are capable of searching directly in sequences of up to 4 GB on typical desktop, while Vmatch is practically limited to

sequences of up to 250 MB on such machines. To overcome this limitation of Vmatch, a possible solution is to partition the long sequence into a number of sequences smaller than 250MB, construct the index for each partition, and perform the search individually in each partition by using its corresponding index. We refer to the former as *direct* search and to the latter as *partitioned* search. We compare the search performance of the three techniques using both direct and partitioned approach for FASST and Trellis, and the partitioned approach for Vmatch, as the only possible option. Note that for the partitioned search we do not include the time needed for partitioning the sequence and for aggregation of found matches. Again, FASST explicitly sorts the query sets, while Trellis and Vmatch are given the sorted query sets.

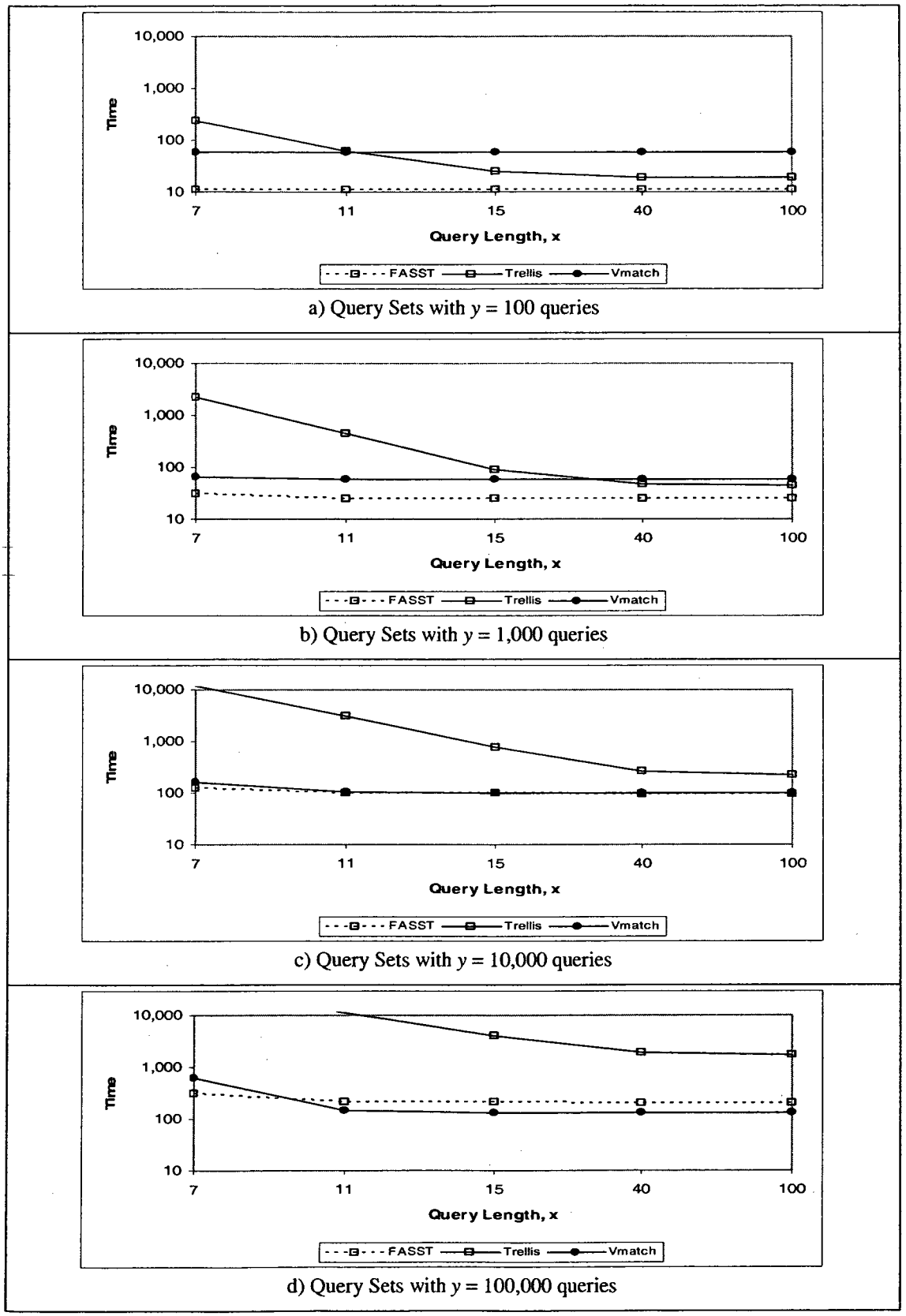
Figure 24 depicts the search times for each query set in *chr1&2* and Figure 25 depicts the search times for each query set in *HG*. The results of these experiments further indicate scalability of FASST with respect to sequence size, which has two important practical implications, as discussed next.

First, FASST is always faster performing direct search compared to partitioned search, explained as follows. Conducting the search directly in *chr1&2* and *HG* returns the same answers we obtain by searching in individual chromosomes. Thus, the number of page accesses to reach all STTD64 leaf nodes in Phase 3 of STEM for these two approaches is about the same. However, direct search leads to a decrease in the number of page accesses in Phase 2. If two (or more) suffixes from different sequences share the same prefix, then this common prefix is represented as a *single path* starting from the root of the ST index built for the concatenated sequences. Thus, during the downwards traversal in Phase 2 (matching query characters to ST edge-labels), we *simultaneously*

search in all concatenated sequences. As a result, the number of random page accesses in Phase 2 decreases proportionally to the number of concatenated sequences, which in turn leads to significant improvement in search times, especially evident for direct search in *HG*. Thus, given a long or a very long sequence, FASST is always faster conducting the search directly, rather than partition the sequence into shorter substrings and search them, even though the 2.7GB *HG* sequence has to be compressed (on-the-fly, using 4 bits to represent a nucleotide) to fit in the 2GB main memory of our desktop. Similarly, given a set of short sequences, it is beneficial for FASST to concatenate them in one very long sequence (if possible), construct the index, and conduct the search directly.

Trellis, the other ST based technique capable of handling directly very long sequences, favors direct search only for longer query sizes. For short queries, Trellis performs better when we search in partitioned sequences (i.e., individual chromosomes). We chose the faster of these two approaches for each query set and used them for illustrating the Trellis performance in Figures 24 and 25.

The second practical implication of the scalability that FASST exhibits with respect to the sequence size is that for long and very long sequences, by performing the search directly, FASST is faster than Vmatch, for all query sets (Figures 24 and 25), except for *q-x-10*^{5}, where Vmatch is about 30% faster. Thus, we conclude that for sequences longer than 250MB, it is beneficial to use FASST and perform direct exact match search – it is up to 5 times faster than Vmatch (for query sets with small number of queries of any length) and up to 100 times faster than Trellis (for query sets with large number of short queries).



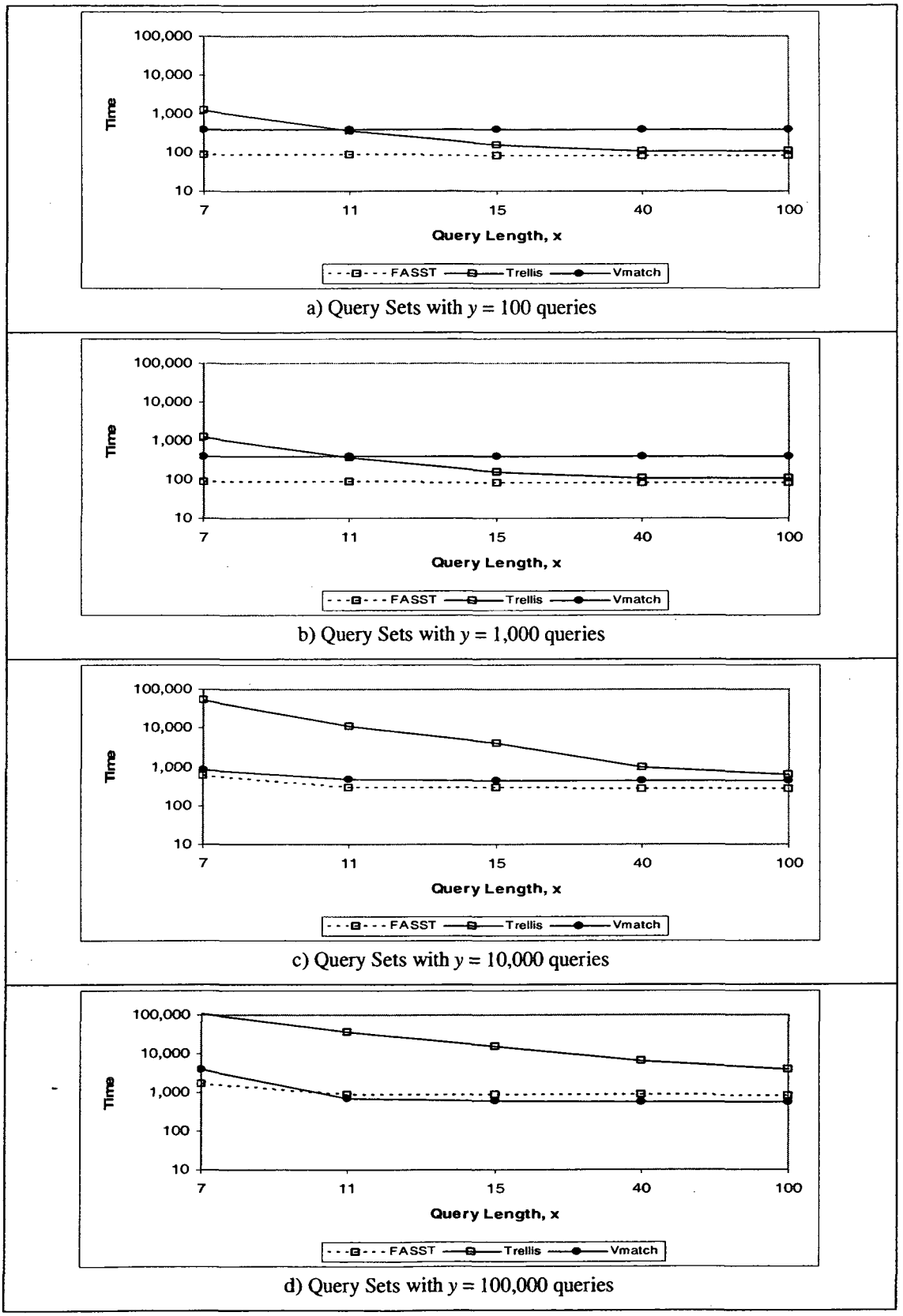
a) Query Sets with $y = 100$ queries

b) Query Sets with $y = 1{,}000$ queries

c) Query Sets with $y = 10{,}000$ queries

d) Query Sets with $y = 100{,}000$ queries

Figure 24. Exact Match Search Times (in seconds), Long Sequences (*chr1&2*, 441MB)



a) Query Sets with y = 100 queries

b) Query Sets with y = 1,000 queries

c) Query Sets with y = 10,000 queries

d) Query Sets with y = 100,000 queries

Figure 25. Exact Match Search Times (in seconds), Very Long Sequences (*HG*, 2.7GB)

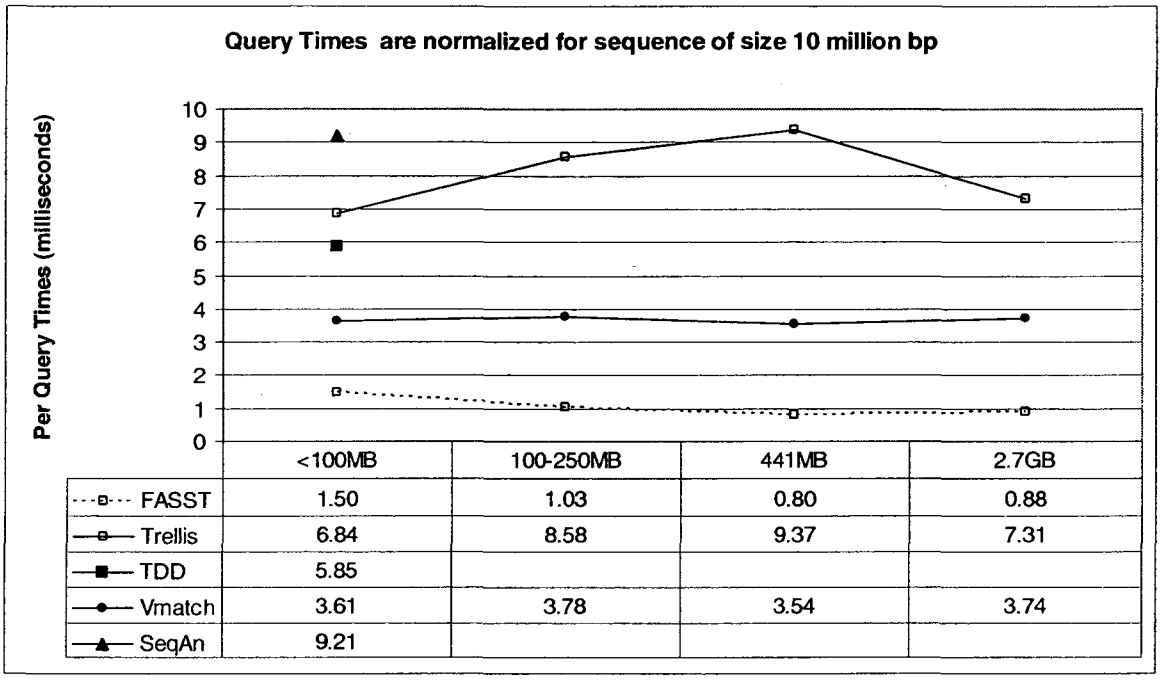
Our results for searching in sequence *chr1&2* also allow an indirect comparison with the LOF-SA technique [Sinha *et al.*, 2008]. For a 400MB DNA sequence and query sets with 10^3 queries, LOF-SA is about 18 times faster than Trellis for query length 10, and about 3 times faster for query lengths 40 and 100 [Sinha *et al.*, 2008]. On our machine, searching in the similarly sized *chr1&2* (441MB) and using the *unsorted* (as done in [Sinha *et al.*, 2008]) query sets *q-11-10*^{3}, *q-40-10*^{3}, and *q-100-10*^{3}, Trellis took 522 seconds, 52 seconds, and 48 seconds, compared to 25 seconds needed by FASST for each query set. Hence, on our machine FASST outperforms Trellis by about the same ratio as LOF-SA does in [Sinha *et al.*, 2008].

The scalability of FASST is further illustrated when performing EMS in long protein sequences. Recall that Trellis does not support protein sequences; Vmatch cannot process sequences of such sizes directly; and while TDD was able to construct the index for the *trembl_old* sequence (840MB), it cannot conduct the search in it, since its exact match search algorithm is memory-based.

We performed the following experiment to investigate the scalability of FASST for searching in long protein sequences. The size of the *trembl_new* is 1.3 GB, and the size of *sprot* is 92 MB, i.e., the ratio between the sizes of the two sequences is 14.5. It might be expected that for the same query sets, searching in *trembl_new* will find about 14.5 times more occurrences. Also, if FASST search times are to scale proportionately with the sequence size, its search times will be about 14.5 times longer. Our experimental results show that while the ratio between the number of occurrences in *trembl_new* and *sprot* matches the size ratio, searching in *trembl_new* is only about 12 times longer for answering the same query sets.

6.4.1.4 Exact Match Summary

We now summarize the results of the experimental evaluation of our proposed FASST and its comparison with existing techniques. For each class of sequence sizes, we computed the search time per query, averaged over query lengths and query set sizes. We normalize these times to a sequence of size 10 million characters and show them in Figure 26.



	<100MB	100-250MB	441MB	2.7GB
FASST	1.50	1.03	0.80	0.88
Trellis	6.84	8.58	9.37	7.31
TDD	5.85			
Vmatch	3.61	3.78	3.54	3.74
SeqAn	9.21			

Figure 26. Normalized Exact Match Search Times by Various Techniques

As can be seen from the figure, FASST in general is about 4 times faster in search than Vmatch, and is about 8 times faster than Trellis. The results also indicate that FASST is significantly faster than TDD and SeqAn, which currently can search in sequences up to 100MB on a desktop computer with 2GB RAM.

On a more detailed level, we make the following observations comparing the exact match search performance of FASST and Vmatch with respect to the query set size. For sequences up to 250MB, FASST is faster for query sets containing up to 1,000 queries,

while Vmatch is faster for larger query sets (Figures 22 and 23). For searching in sequences longer than 250MB, using direct search FASST significantly outperforms Vmatch for most query set sizes, except for very large ones, for which Vmatch has a slight advantage (Figures 24 and 25).

For further comparison of these two techniques, we studied their performance for a more involved search task, the approximate k-mismatch search, discussed next.

6.4.2 K-mismatch Search (KMS)

In this section, we study the performance of our STKM algorithm which implements the k-mismatch search functionality in FASST and compare it to Vmatch solution, the only one among the considered techniques that support such functionality at this time. TDD has a k-mismatch search algorithm; however it produces run time errors. We start by explaining how the query sets used in our experiments are generated. We next present the search times for the considered techniques organized according to sequence sizes.

We use three query sets: the query set *q-100-10*^{3}, used in the exact match search experiments, and two additional query sets, *q-500-10*^{3} and *q-1000-10*^{3}, generated in the same manner as described in Section 6.4.1. For k, we consider the values 1, 5, and 10; 10 is the largest k value accepted by Vmatch. The query sizes are representative of the length of Expressed Sequence Tags (EST), short sub-sequences of a transcribed cDNA sequence, whose presence or absence in a new DNA sequence helps in gene discovery and gene sequence determination. In evaluating the k-mismatch performance of FASST and Vmatch, we chose to fix the number of queries in each query set to 10^3 queries, for the following reasons.

First, for query sets with 10^3 queries, FASST and Vmatch exhibit quite similar exact match search performance in sequences up to 250MB (Figures 22.b and 23.b). Thus, using query sets with 10^3 queries provides FASST and Vmatch with a fair starting point for their k-mismatch search time comparison, unbiased by their performance for exact match search.

Second, while the number of queries in the query sets may look insufficient for reliable analysis, this is not really true. Even for the shortest query length of 100 nucleotides and the smallest possible value of $k \leq 1$, posing a query set with 10^3 queries corresponds to posing more than 300,000 exact match search queries, if one were to take the brute force approach suggested in [Gusfield, 1997]. Although STKM avoids this complexity blowup, the amount of work required for the query sets and the k values considered still remain significant.

In our k-mismatch search experiments, we classify the DNA sequences as chromosome size sequences (up to 250MB) and genome size sequences (2.7GB).

6.4.2.1 KMS Comparison in Chromosome Sequences (< 250MB)

Table 5 shows the measured KMS times (in seconds) for k-mismatch search in chromosome size sequences (up to 250MB). Comparing the two techniques for all the 9 possible pairs of query sets and k values considered, we note that FASST outperforms Vmatch in all these cases. The total search time was 10,912 second (182 minutes) using FASST, and was 19,523 seconds (325 min) using Vmatch, and hence FASST is on average about 2 times faster than Vmatch on k-mismatch search.

Table 5. FASST vs. Vmatch on KMS for DNA sequences up to 250MB

Query Set	$k = 1$		$k = 5$		$k = 10$	
	FASST	**Vmatch**	**FASST**	**Vmatch**	**FASST**	**Vmatch**
q-100-10³	**591**	1,612	**1,251**	1,818	**1,893**	2,727
q-500-10³	**596**	2,271	**1,237**	2,214	**1,749**	2,195
q-1000-10³	**592**	2,206	**1,252**	2,228	**1,751**	2,252

A closer look at the results in Table 5 shows that, with respect to all considered k values, the speedup obtained by FASST range from 1.3 (k = 10) to 3.4 (k = 1) compared to Vmatch. With respect to the query size and FASST is on average 1.8 times faster than Vmatch. The smallest performance gap between the two techniques occurs for query set *q-1000-10³* and k = 10, where FASST is only 1.3 times faster, while the largest performance gap is for query set *q-500-10³* and k = 1, where FASST is 3.8 times faster, as illustrated in Figure 27.

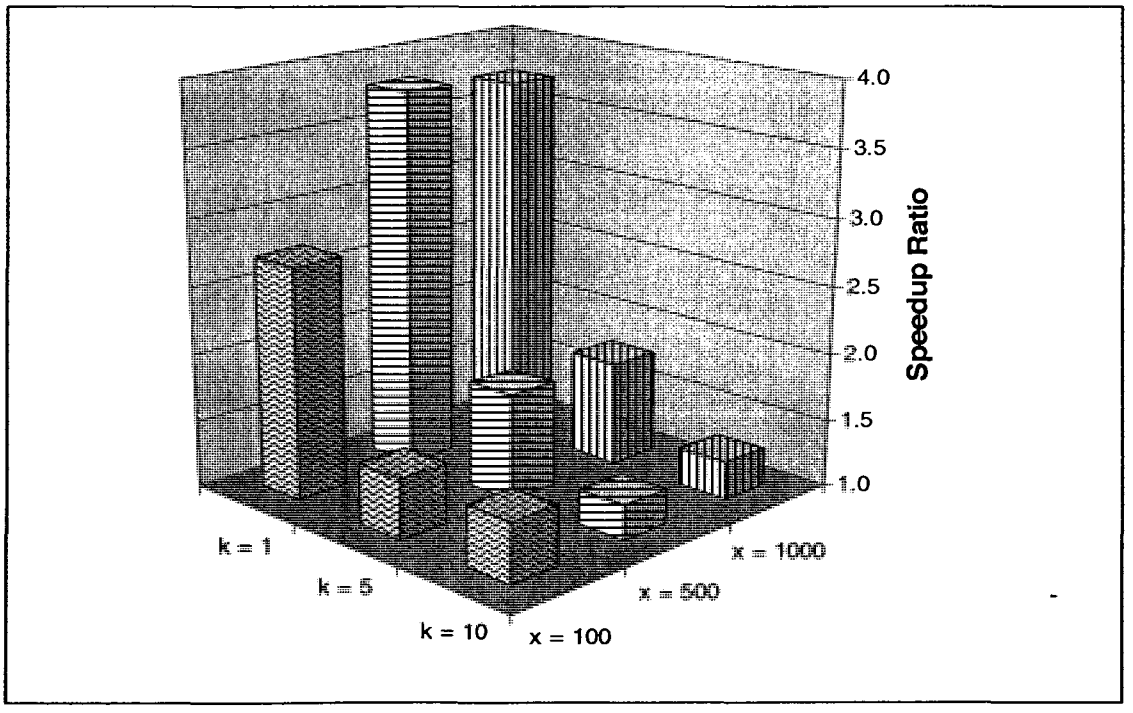


Figure 27. FASST/Vmatch Speedup Ratio for KMS (up to 250MB)

Based on the results presented in Table 5 and Figure 27, we may conclude that for chromosome size sequences (up to 250MB), FASST provides faster k-mismatch search

than Vmatch. The question of how efficient is FASST in performing k-mismatch search directly in long, genome size sequences is addressed next.

6.4.2.2 KMS Comparison in Genome Sequence (2.7GB)

Next, we investigate the k-mismatch search performance of FASST in genome size sequences, using the *HG* sequence (2.7GB), and evaluate the benefits of conducting the KMS directly, using FASST, compared to performing the search at chromosome level using either FASST or Vmatch.

Table 6 reports the observed search times (in seconds) for conducting the KMS search directly in *HG* using FASST, denoted FASST(dir), and in the individual chromosomes using FASST and Vmatch, labeled as FASST(part) and Vmatch(part), respectively.

Table 6. KMS for DNA sequences longer than 250MB

Query Set	$k = 1$			$k = 5$			$k = 10$		
	FASST (dir)	FASST (part)	Vmatch (part)	FASST (dir)	FASST (part)	Vmatch (part)	FASST (dir)	FASST (part)	Vmatch (part)
q-100-10^3	**143**	591	1,612	**265**	1,251	1,818	**620**	1,893	2,727
q-500-10^3	**144**	596	2,271	**260**	1,237	2,214	**399**	1,749	2,195
q-1000-10^3	**142**	592	2,206	**259**	1,252	2,228	**398**	1,751	2,252

Our results of performing KMS in long DNA sequences are consistent with those obtained for performing EMS in such sequences. As can be seen from the above table, FASST exhibits good scalability with respect to the sequence size. Our results show that the direct FASST k-mismatch search in *HG* is on average 4 times faster than the partitioned FASST approach, for the same reason as for exact match search. Further, given a genome size sequence, FASST provides 5 to 14 times faster k-mismatch by searching directly in the long sequence, compared to Vmatch, which must search in the shorter partitions.

Overall, our results in Figure 25 and Table 6 suggest that given a genome size sequence, it is *not* beneficial to partition such sequence into shorter sequences and use the existing Vmatch tool for performing exact match or k-mismatch search. For long sequences, our experimental results indicate that it is advantageous (up to 5 times faster exact match search and up to 14 times faster k-mismatch search) to create the HST index for the entire long sequence and use FASST to perform exact match and k-mismatch search directly in the long sequence.

Next, we evaluate the performance of FASST for a practical bioinformatics application, structured motif search, and compare its performance to the best known existing solution.

6.4.3 Structured Motif Search (SMS)

In this section, we study the performance of our EMOS algorithm which implements the structured motif search functionality in FASST. Since Vmatch does not support motif search, we compare FASST performance to the best known alternative, SMOTIF1 [Zhang and Zaki, 2006]. We start by describing the query sets used in our experiments. We next present and analyze the search times for the considered techniques.

In our experiments we used two sets of queries. The first set is the same collection of randomly generated structured motifs used in [Zhang and Zaki, 2006], which was provided to us by the authors. The set contains 100 random structured motifs over the IUPAC alphabet. Each structured motif consists of 3 to 8 simple motifs of length between 5 and 10 symbols. The number of simple motifs and their lengths are selected uniformly at random within these ranges. The gaps between simple motifs are chosen as a random subinterval of [-5, 100]. Recall that negative values for the gap size allow for partially

overlapping simple motifs. The second set contains the four real-life structured motifs used in [Zhang and Zaki, 2006], obtained by a multiple alignment of 36 *A. thaliana* LTR retrotransposons. The motifs are shown in Figure 28 (where *ZZ* stands for Zhang & Zaki).

ZZ1 = HNGTNYDNHDNBTNNDNA[0,3]YNHTNYRHGGNBTNAR[0,2]ARDBNBH
ZZ2 = TNVRNKAYNKNVVNDV[9,11]HNRR[6,8]YDNNVNNV[9,13]HB[4,5]TNNNNRBNYDBDNNRR
ZZ3 = DNNNNDRYW[2,5]DS[6,7]HMM[1,2]TNDB
ZZ4 = DBNNNND[48,102]KRRYMYNNNNMRNHYNDVNYAYVH[7,10]VNNNYNNND[34,63]
WD[2,8]KNNH[3,5]VNDDRNNNNNNNHVNNNNNNNHHH

Figure 28. Real-life structured motifs [Zhang and Zaki, 2006]

In our experiments we used the source code of SMOTIF1 available at [SMOTIF, 2007]. Its current implementation is limited to processing a single structured motif query at a time. Although FASST accepts a set of structured motifs as input, in our experiments we pose only a single query at a time to both programs. This makes the comparison fair to SMOTIF1, since otherwise FASST is much faster on a set of queries. The reported search times are based on the assumption that the HST index has been already constructed and available to FASST. Later in this section, we revisit this assumption.

6.4.3.1 SMS Comparison: FASST vs. SMOTIF1

Our first experiment compares the performance of FASST and SMOTIF1 using the collection of 100 synthetic structured motifs. Table 7 reports the search times (in seconds), accumulated for all 100 queries. Our results indicate that FASST is 5 to 6 times faster than SMOTIF1, for the reasons explained in Section 6.4.3.2.

Table 7. FASST vs. SMOTIF1 on SMS for 100 random structured motifs

Sequence	Sequence Size	SMOTIF1	FASST	Speedup
chrY	26 Mb	422	**88**	4.8
chr20	60 Mb	1,072	**184**	5.8
chr10	132 Mb	2,205	**371**	5.9
chr2	238 Mb	3,859	**645**	6.0

Our second experiment compares the performance of the two techniques using the 4 real-life structured motifs and the search times are presented in Figure 29.

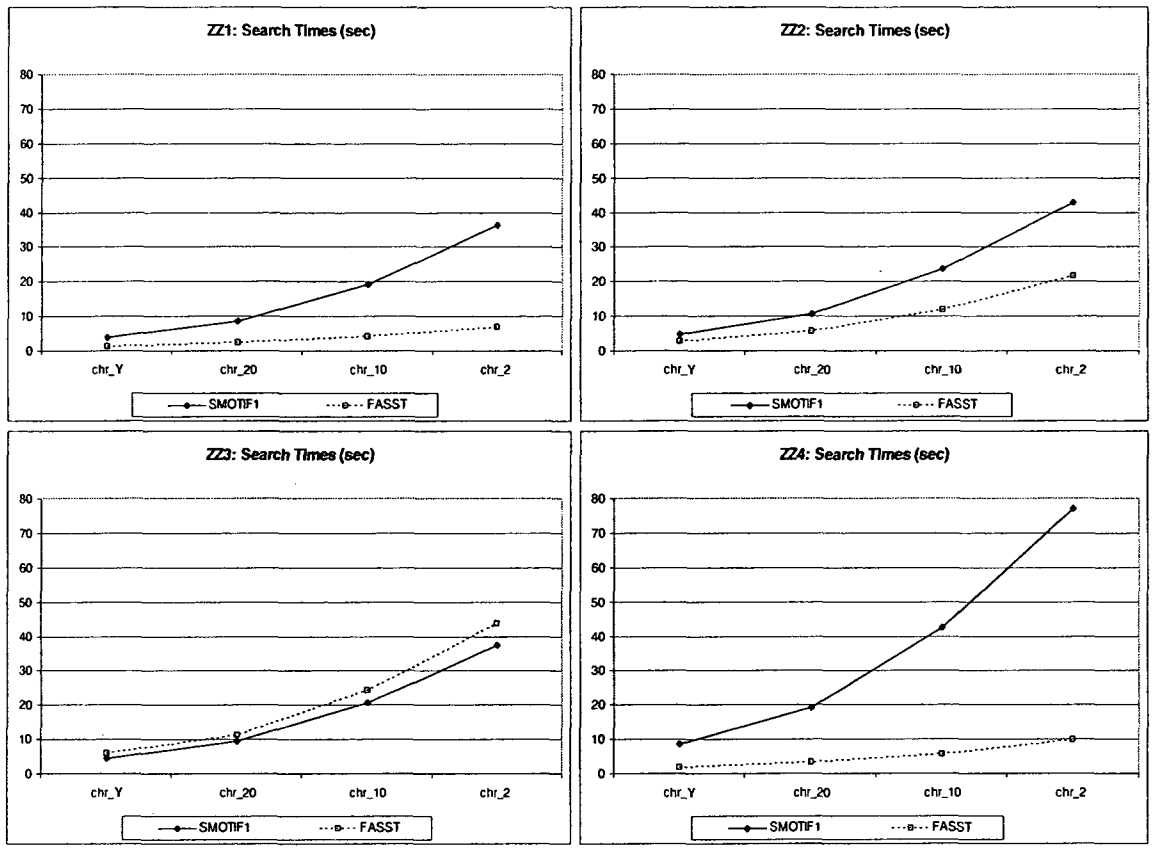


Figure 29. FASST vs. SMOTIF1 on SMS for real-life structured motifs

Again, for structured motifs with practical selectivity, FASST significantly outperforms SMOTIF1, providing up to 8 times faster search (i.e., *ZZ4* in *chr2*). For *ZZ3*, due to the low selectivity power of all its simple motifs (there are almost 4 million occurrences of *ZZ3* in *chr2*), we cannot take advantage of selecting a suitable simple motif M_a which would reduce the work done in Step 3 of EMOS (Figure 18). In this case, performance of FASST is similar to SMOTIF1.

It also should be noted that the search time advantage of FASST over SMOTIF1 increases with longer sequence sizes. Next, we analyze the behavior of the proposed

EMOS algorithm which implements the structured motif search in FASST, focusing on its sources of efficiency.

6.4.3.2 Sources of Improvement

There are two major sources of efficiency for EMOS. First, our approach of finding exact matches for a single simple motif and then aligning the structured motif with the sequence and performing character comparisons is more efficient compared to the approach taken by SMOTIF1, which is based on extracting the full post-lists for all motif symbols and then performing positional joins on them. Second, the heuristic used for selecting a suitable simple motif, although not optimal, significantly reduces the amount of work done by our algorithm, resulting in improved search time. Next, we discuss these sources of efficiency in more detail.

To evaluate the first source of improvement, in our third set of experiments, we modified EMOS so that a random simple motif is selected as M_a and refer to this variation as *EMOS (random)*. This makes its performance independent of how suitable the selected M_a is. The measured search times (in seconds) are presented in Table 8.

Table 8. EMOS (random) vs. SMOTIF1 on SMS for 100 random structured motifs

Sequence	SMOTIF1	EMOS (random)	Speedup
chrY	422	**174**	2.4
chr20	1,072	**410**	2.6
chr10	2,205	**935**	2.4
chr2	3,859	**1,728**	2.2

As can be seen from the results, even when selecting M_a at random, our approach is about twice faster than SMOTIF1. This is explained by the fact that in the first step, SMOTIF1 considers all possible locations in the sequence, i.e., the *full* post-lists of all structured motif (*SM*) symbols. In the second step, it explores these post-lists by

performing positional joins. On the other hand, by performing an initial exact match search for a single simple motif, *EMOS* (*random*) greatly reduces the number of sequence locations (usually less than 10% of the sequence size) that has to be further examined in Step 3 of EMOS by aligning the *SM* with *S* at these anchor locations.

The second source of the improvement provided by EMOS is based on selecting a *suitable* simple motif as M_a. The goal is to choose M_a in such a way that the number of anchor positions that has to be examined in Step 3 is minimal. We choose M_a by computing the selectivity powers of all simple motifs as discussed in Section 5.3. Table 9 shows the search times (in seconds) and the improvement obtained by selecting and using a suitable M_a as done by FASST, compared to using a random simple motif as M_a, as in *EMOS* (*random*). Our experimental results indicate that by selecting a suitable M_a, the number of sequence locations that have to be examined in Step 3 of EMOS is further reduced by a factor of 10 (i.e., anchor locations are now less than 1% of the sequence size). As a result, an additional search time improvement of 2 to 2.7 times is obtained.

Table 9. EMOS for 100 structured motifs: Random vs. Suitable M_a

Sequence	EMOS (random)	EMOS	Speedup
chrY	174	**88**	2.0
chr20	410	**184**	2.2
chr10	935	**371**	2.5
chr2	1,728	**645**	2.7

Recall that our heuristic for selecting a suitable M_a is based on the assumption of uniform distribution of the DNA bases. However, in real-life DNA sequences the base distribution is not strictly uniform. An alternative is to read the sequence content and obtain the actual base distribution. In Table 10, we present the uniform and actual distribution of the IUPAC symbols in the 24 human chromosomes. The question is: How

using actual base distribution affects the accuracy of estimated number of exact matches of M_a in S and is it beneficial to take this approach, in terms of the overall search time?

Table 10. Distribution of IUPAC Symbols in Human Chromosomes

Symbol	A	C	G	T	R	Y	K	M	S	W	B	D	H	V	N
Uniform	25%	25%	25%	25%	50%	50%	50%	50%	50%	50%	75%	75%	75%	75%	100%
Actual	29%	21%	21%	29%	50%	50%	50%	50%	42%	58%	71%	79%	79%	71%	100%

In our last set of experiments, using the 100 random structured motifs, we study the accuracy of the estimator for these two alternative approaches. We compute the estimation percentage error defined as: E = (*num_act* – *num_est*) / *num_est*, where *num_est* is the number of matches of M_a in S estimated by the heuristic and *num_act* is the actual number of matches found by searching the sequence.

For both approaches, the sample distribution of the estimation error is approximately Gaussian. Assuming uniform distribution, our heuristic slightly overestimates the number of actual exact matches (mean error = -4%). However, the standard deviation of the error is relatively high (σ = 57%), reflecting that in some cases the estimator performs poorly. On the other hand, using the actual IUPAC distribution, the heuristics selects different M_a for 14 out of the 100 structured motifs (for this approach, mean error = -8%, σ = 43%). However, the slight improvement in estimation accuracy achieved by using the actual distribution is not sufficient to offset the additional time spent for reading the sequence content and overall EMOS is slower taking this approach.

Now, we revisit our assumption that HST index has been already created and available to EMOS. This is a fair assumption, since the content of biological sequences is relatively stable and there are numerous other applications (such as EMS, KMS, etc.) that benefit from an index which already exists. However, if the index has not been created in

advance, it can be constructed for around 900 seconds for *chr2*, for example. The search times for *chr2* in Table 7 indicate that the cost of creating the HST index will be amortized after performing about 30 structured motif searches, if posing each motif query individually.

Last, we remark that on average 70% of the EMOS search time is spent loading the data sequence from disk to memory. Of the remaining 30%, selecting a suitable simple motif (Step 1) takes less than 1%, using the HST index to obtain all exact match occurrences of the anchor motif (Step 2) requires around 10%, and extending the structured motif at the anchor positions (Step 3) accounts for the last 20%. The search time distribution explains why EMOS is much faster when searching in a particular sequence for a set of motif queries, as opposed to a single query at a time. For example, using EMOS to search for our query set of 100 random motifs is above 20 times faster compared to searching for each motif individually, as done in SMOTIF1. The search time distribution also implies that a more efficient implementation of Step 3 will further improve the search time of EMOS. We plan to investigate this issue, by considering alternative approaches, such as the positional join approach of SMOTIF1.

6.4.4 Supermaximal Repeats Search

In a recent work, [Lian *et al.*, 2008], we studied applicability of the HST index for finding supermaximal repeats (SMR) in large DNA sequences. For this, we proposed an algorithm, called SMR, based on an auxiliary index structure (POL), which is derived from HST and replaces it for the needs of SMR application. The results of our experiments using the 24 human chromosomes indicated that SMR outperforms Vmatch. In searching for supermaximal repeats of size at least 10 bases, SMR is twice faster than

Vmatch; for a minimum length of 25 bases, SMR is 7 times faster; and for repeats of length at least 200, SMR is about an order of magnitude faster than Vmatch. The SMR problem is another application which HST efficiently and effectively supports.

7 Conclusions and Future Work

The amount of publicly available biological sequence data, represented as long strings over the DNA and protein alphabets, is of considerable size and grows at phenomenal rate. The goal of this thesis was on providing efficient, scalable, and versatile processing of and searching in such sequences on regular desktop computers. To achieve this goal, we focused on appropriate indexing techniques tailored to the specifics of the domain.

Recently, suffix tree (ST) and suffix array (SA) received considerable attention as suitable data structures for indexing sequences in our context. However, existing solutions provide either scalability (i.e., the suffix tree based TDD [Tian *et al.*, 2005] and Trellis [Phoophakdee and Zaki, 2007]) or efficiency (i.e., the enhanced suffix array based Vmatch [Vmatch, 2007]), but not both. Further, some of the proposed techniques require expensive computational resources.

Our work addresses these issues and investigates, both theoretically and experimentally, appropriate models for indexing biological sequence data and required search techniques. The result of our research is a software package, called FASST (Fast And Scalable Search Tool), designed for regular desktop computers and consisting of three major parts – HST, a novel ST based index representation for sequence data; a disk-based index construction algorithm, and a set of disk-based search applications that use the index to provide solutions to various search tasks.

Next, we highlight the properties of the proposed HST index and summarize the results of our numerous experiments with real-life data, which provide the basis of our conclusions.

7.1 Conclusions

The HST index (Halachev, Shiri, Thamildurai) is a two-level index structure, which combines a lookup table (LT) and a suffix tree (STTD64), briefly described next.

Each ST node is represented as a single 64-bit record in our STTD64 representation, regardless of being a branch or a leaf node. The STTD64 index for a sequence S is implemented as a linear array of these records. Note that our design choice for using 64 bits for representing the ST nodes does not imply a 64-bit architecture, although it would benefit from it.

The main difference between STTD64, on one hand, and *wotd*, TDD, and Trellis suffix tree representations, on the other, is that in STTD64 we store the depth of each leaf node. Having this information available eliminates the need to traverse the ST for computing the starting positions of suffixes, thus significantly improving the locality of reference exhibited by the search algorithms using the index.

The other component of HST, the LT index, contains the following information. Given a fixed parameter p (a small positive integer), for each unique string l with length p over the sequence alphabet, we store a pointer to the STTD64 node which represents l. Thus, the small, memory-resident LT serves as an index to the large, disk-resident STTD64, and further improves the locality of reference of the search algorithms using the HST index by providing basis for implementation of an efficient buffering strategy.

Our proposed HST index overcomes the 1GB limit of the *wotd* representation. HST can handle longer sequences, with a theoretical limit of 4 GB, thus matching the capabilities of TDD and Trellis. In [Halachev *et al.*, 2009], we have shown HST effectively constructed in 16 hours for the entire human genome (approximately 2.7 GB)

on a regular desktop computer with 2 GB RAM, which time is similar to the index construction times for TDD and Trellis. HST and TDD indexes have similar storage requirements, while Trellis is two times larger. However, the main advantage of HST over TDD and Trellis is its suitability for disk-based computation – the search applications using HST exhibit significantly improved disk I/O behavior, which translates to faster search times.

One limitation of HST is that it cannot handle sequences containing exactly repeated substrings of size longer than 2^{30} (1 billion) bases. However, this restriction is not likely to be an issue when dealing with real-life biological data.

We have conducted numerous experiments with different practical parameters using real-life data sequences to evaluate our proposed indexing technique and to compare its performance to existing solutions. In essence, our comparison study answers the following question: Given a biological sequence of certain size, a particular query set, and a search task to be performed, which is the most efficient technique?

The results of our experimental study indicate that our proposed HST indexing technique for biological sequences provides:

1) Ability to handle both DNA and protein sequence data;
2) Reasonable index construction times, comparable to the most efficient disk-based ST construction techniques, TDD and Trellis;
3) Ability to index sequences up to 4GB, which matches the TDD and Trellis limit and is an order of magnitude larger than the limit for the memory-based Vmatch;
4) Storage requirements similar to TDD and Vmatch, and twice smaller than Trellis ;

5) Efficient and scalable support for various search applications - due to the improved locality of reference, the HST based exact match (Section 6.4.1), k-mismatch, and structured motif search algorithms outperform the existing state-of-the-art solutions;
6) Ability to conduct various types of search tasks in genome-scale biological sequences on typical desktop computers (e.g., 2 GB RAM, single processor, single hard disk, 32-bit architecture);

Based on these results, we make the following conclusions.

First, compared to Trellis, the best-known suffix tree based solution for handling long DNA sequences (up to 4GB), our proposal provides significantly faster search performance, while having comparable index construction cost. Compared to TDD, another suffix tree based indexing technique, which can handle both DNA and protein sequences, our proposal provides considerably faster and more scalable search, again at comparable index construction cost. Thus, we conclude that among the existing suffix tree based solutions, the HST indexing technique is the most desirable.

Second, compared to Vmatch, the best-known suffix array based solution, our proposal exhibits two to three times slower index construction, and overall comparable search performance for the sequences Vmatch can handle. The main advantage of the disk-based HST is its capability to process and search directly in DNA and protein sequences up to 4GB on a typical desktop computer, while with the same computational resources the memory-based Vmatch is limited to sequences up to 250MB. Our

experimental results indicate that given a long sequence, it is *not* beneficial to partition it into shorter substrings and perform the search using Vmatch. In fact, performing exact and k-mismatch search directly into a long sequence using its HST index is several times faster compared to Vmatch, which must search into the partitions. Thus, we conclude that due to its efficient support for wider range of sequence sizes, the HST indexing technique is more amenable to the bioinformatics domain.

Third, in addition to the support the HST index provides to the exact match and k-mismatch search problems, we present efficient HST based solutions to other, more involved bioinformatics search tasks, such as structured motif search and finding supermaximal repeats. We view these results as a strong indication of the versatility of the proposed indexing technique.

In summary, we conclude that the proposed suffix tree based HST indexing technique supports efficient and scalable search in DNA and protein sequences and has a great potential in development of an appropriate management system for biological sequences.

7.2 Future Work

Motivated by the great potential exhibited by HST as a suitable index for biological sequences, our work can be extended in several directions, as follows:

- Improve the time performance of the HST construction algorithm by developing more efficient sorting algorithm for finding the longest common prefix of the suffixes in the current partition;
- Develop a HST based solution for k-difference search, which together with exact match and k-mismatch search form a core of search tasks. Solving this problem

may require providing an efficient solution to the lowest common ancestor (*lca*) problem, a solution for which must be included in any string/biological sequence processing tool [Gusfield, 1997];

- Use the solutions for exact, k-mismatch, and k-difference search tasks as building blocks for more involved bioinformatics exact and/or approximate search applications, such as:
    - structured motif extraction, which allows for finding composite transcription factors that regulate a gene;
    - finding longest common substring/subsequence of two or more sequences, for which suffix tree indexes are extremely suitable [Gusfield, 1997], and which is at the heart of alignment techniques;
    - local/global, pairwise/multiple alignments of biological sequences.
- While in this work we focused on typical desktop computers, our suffix tree based indexing technique could benefit from a possible extension to parallel/distributed computation. Construction of a suffix tree index seems ideally suited for parallelization ([Tian *et al.*, 2005], [Phoophakdee and Zaki, 2007]). The idea is to divide all sequence suffixes into several partitions based on their common prefix (of certain length) and for each such partition build its corresponding subtree, which is independent of the other subtrees. Then, the search would be done using the subtrees which implicitly constitute the whole suffix tree. For this, the content of the LT table should be augmented to include the information about the physical location (e.g., which hard disk/which machine) for each particular index subtree. Further, for more advances applications there is a good potential of pipelining the

search algorithm. For example, the structured motif search algorithm described in Section 5.3 would benefit from performing the verification step (Step 3) for each promising location as soon as such location is computed (Step 2), rather than executing the verification step after all promising locations are determined in Step 2. Extending our proposed indexing technique to parallel/distributed computing environment would require further investigation of proper approaches for data replication, resource allocation, and efficient communication protocol.

Bibliography

[**Abouelhoda et al., 2004**] Abouelhoda, M., Kurtz, S., and Ohlebusch, E. *Replacing suffix trees with enhanced suffix arrays.* J. Discrete Algorithms, 2(1): 53-86, 2004

[**Altschul and Gish, 1990**] Altschul, S. and Gish, W. *Basic local alignment search tool.* J. Molecular Biology, 215(3): 403-410, 1990

[**Altschul et al., 1997**] Altschul, S. F., Madden, T. L., Schaeer, 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.* Nucleic Acids Research, 25(17): 3389-3402, 1997

[**BLAST, 2008**] http://www.ncbi.nlm.nih.gov/blast

[**Andersson and Nilsson, 1995**] Andersson, A. and Nilsson, S. *Efficient implementation of suffix tree.* Softw. Pract. Exp., 25(2): 129-141, 1995

[**Apostolico, 1985**] Apostolico, A. *The myriad virtues of subword trees.* Combinatorial Algorithms on Words. Apostolico, A. and Galil, Z. (Eds.), NATO ASI Series F: Computer and System Sciences, Springer-Verlag, New York, pp. 85–96, 1985

[**Apostolico and Giancarlo, 1986**] Apostolico, A. and Giancarlo, R. *The Boyer-Moore-Galil string searching strategies revisited.* SIAM J. Computing 15(1): 98-105, 1986

[**Baeza-Yates and Navarro, 1999**] Baeza-Yates, R.A. and Navarro, G. *Faster approximate string matching.* Algorithmica, 23(2): 127-158, 1999

[**Baeza-Yates et al., 1996**] Baeza-Yates, R.A., Barbosa, E.F., and Ziviani, N. *Hierarchies of indices for text searching.* Information Systems 21(6), pp. 497-514, 1996

[**Bedathur and Haritsa, 2004**] Bedathur, S. and Haritsa, J. *Engineering a fast online persistent suffix tree construction.* Proc. of ICDE`04, pp. 720-731, USA, 2004

[**Boyer and Moore, 1977**] Boyer, R.S. and Moore, J.S. *A fast string searching algorithm.* Communications of the ACM. 20: 762-772, 1977

[**Bray et al., 2003**] Bray, N., Dubchak, I., and Pachter, L. *AVID: A global alignment program.* Genome Research, 13(1): 97–102, 2003

[**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 (QUASAR).* Proc. of RECOMB`99, pp. 77-83, France, 1999

[**Cameron, 2006**] Cameron, M. *Efficient Homology Search for Genomic Sequence Databases.* PhD Thesis, RMIT University, Melbourne, Victoria, Australia, 2006

[**Carvalho et al., 2004**] Carvalho, A., Freitas, A., Oliveira, A., and Sagot, M. *Efficient extraction of structured motifs using box-links.* Proc. of SPIRE`04, pp. 267-278, Italy, 2004

[**Chang and Lawler, 1994**] Chang, W. and Lawler, E. *Sublinear approximate string matching and biological applications.* Algorithmica, 12(4/5): 327-344, 1994

[**Chen, 2004**] Chen, X. *A framework for comparing homology search techniques.* Master's thesis, RMIT University, Melbourne, Victoria, Australia, 2004

[**Cheung et al., 2005**] Cheung, C., Yu, J., and Lu, H. *Constructing suffix tree for gigabyte sequences with megabyte memory.* IEEE Transactions on Knowledge and Data Engineering, 17(1), pp. 90–105, 2005

[**Cormen et al., 1990**] Cormen, T. H., Leiserson, C. E., and Rivest, R.L. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, USA, 1990

[**Crauser and Ferragina, 2002**] Crauser, A. and Ferragina, P. *A theoretical and experimental study on the construction of suffix arrays in external memory.* Algorithmica, 32(1): 1-35, 2002

[**Darling and Feng, 2002**] Darling, A. and Feng, W. *mpiBLAST: Parallelization of BLAST for computational clusters.* Proc. of Supercomputing`02, Baltimore, USA, 2002

[**dbEST, 2007**] dbEST Database. http://www.ncbi.nlm.nih.gov/dbEST. Cited July, 07

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

[**Delcher et al., 2002**] Delcher, A., Phillippy, A., Carlton, J., and Salzberg, S. *Fast algorithms for large-scale genome alignment and comparison.* Nucleic Acids Research, 30(11): 2478–2483, 2002

[**Dementiev et al., 2005**] Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. *Better external memory suffix array construction.* Proc. of ALENEX`05, pp. 86-97, 2005

[**Dorohonceanu and Nevill-Manning, 2000**] Dorohonceanu, B. and Nevill-Manning, C.G. *Accelerating protein classification using suffix trees.* Proc.of ISMB`00, pp. 128-133, USA, 2000

[**ExPASy, 2006**] ftp://ca.expasy.org/databases/swiss-prot/release. Cited May, 2006

[**ExPASy, 2007**] ftp://ca.expasy.org/databases/swiss-prot/release. Cited April, 2007

[**FASST, 2008**] http://sepehr.cs.concordia.ca/

[**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*". J. of the ACM, 46(2): 236-280, 1999

[**Feschotte et al., 2002**] Feschotte, C., Jiang, N., Wessler, S. *Plant transposable elements: where genetics meets genomics.* Nature Review Genetics 3(5), 329-341, 2002

[**GenBank, 2007a**] GenBank Database. http://www.ncbi.nlm.nih.gov/Genbank. Cited August, 07

[**GenBank, 2007b**] GenBank Size. ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt. Cited August, 07

[**Giegerich et al., 2003**] Giegerich, R., Kurtz, S., and Stoye, J. *Efficient implementation of lazy suffix trees.* Softw. Pract. Exper.; 33(11): 1035-1049, 2003

[**Giladi et al., 2000**] Giladi, E., Walker, M.G., Wang, J.Z., and Volkmuth, W. *SST: An algorithm for searching sequence databases in time proportional to the logarithm of the database size.* Proc. of RECOMB`00, poster, Japan, 2000

[**Gusfield, 1997**] Gusfield, D. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997

[**Gusfield and Stoye, 2004**] Gusfield, D. and Stoye, J. *Linear time algorithms for finding and representing all the tandem repeats in a string.* J. Computer and System Sciences, 69(4): 525-546, 2004

[**Halachev et al., 2005**] Halachev, M., Shiri, N., and Thamildurai, A. *Exact Match Search in Sequence Data Using Suffix Trees.* Proc. of CIKM`05, pp.123-130, Germany, 2005

[**Halachev et al., 2007**] Halachev, M., Shiri, N., and Thamildurai, A. *Efficient and Scalable Indexing Techniques for Biological Sequence Data.* Proc. of BIRD`07 - LNBI vol. 4414, Hochreiter, S. and Wagner, R. (Eds.), Springer Verlag, pp. 464-479, Germany, 2007

[**Halachev and Shiri, 2008**] Halachev, M., and Shiri, N. *Fast Structured Motif Search in DNA Sequences.* Proc. of BIRD`08 - CCIS vol.13, Elloumi, M. et al. (Eds.), Springer Verlag, pp. 58-73, Austria, 2008

[**Halachev et al., 2009**] Halachev, M., Shiri, N., and Thamildurai, A. *Efficient and Scalable Search in DNA Sequences.* Submitted to the TCBB Journal, January 2009.

[**Höhl et al., 2002**] Höhl, M., Kurtz, S., and Ohlebusch, E. *Efficient multiple genome alignment.* Bioinformatics 18 (supplement 1): 312–320, 2002

[**Horspool, 1980**] Horspool, R. N. *Practical fast searching in strings.* Softw. Pract. Exp., 10(6):501-506, 1980

[**Hunt, 2003**] Hunt, E. *The Suffix Sequoia Index for Approximate String Matching.* Technical report, http://www.dcs.gla.ac.uk/~ela/TR-2003-135.ps.gz, 2003

[**Hunt et al., 2002**] Hunt, E., Atkinson, M.P., and Irving, R.W. *Database indexing for large DNA and protein sequence collections.* VLDB Journal, 11: 256-271, 2002

[**Irving and Love, 2003**] Irving, R.W. and Love, L. *The Suffix Binary Search Tree and Suffix AVL Tree.* J. of Discrete Algorithms, 1: 387–408, 2003

[**Jagadish et al., 2000**] Jagadish, H. V., Koudas, N., and Srivastava, D. *On effective multi-dimensional indexing for strings.* ACM SIGMOD Record. 29(2): 403-414, 2000

[**Jurka et al., 2005**] Jurka, J., Kapitonov, V., Pavlicek, A., Klonowski, P., Kohany, O., and Walichiewicz, J. *Repbase Update, a database of eukaryotic repetitive elements.* Cytogenet Genome Res. 110(1-4): 462-467, 2005

[**Karp and Rabin, 1987**] Karp, R. and Rabin, M. *Efficient randomized pattern matching algorithms.* IBM J. Res. Development, 31(2): 249-260, 1987

[**Knuth et al., 1977**] Knuth, D.E., Morris (Jr), J.H., and Pratt, V.R. *Fast pattern matching in strings.* SIAM J. Computing 6(1): 323-350, 1977

[**Kurtz, 1999**] Kurtz, S. *Reducing the space requirement of suffix trees.* Softw. Pract. Exp., 29(13): 1149-1171, 1999

[**Kurtz and Schleiermacher, 1999**] Kurtz, S. and Schleiermacher, C. *REPuter: fast computation of maximal repeats in complete genomes.* Bioinformatics 15: 426-427, 1999

[**Levenshtein, 1966**] Levenshtein V. I. *Binary codes capable of correcting, deletions, insertions and reversals.* Soviet Phys. Dokd, 10: 707-710, 1966

[**Lian et al., 2008**] Lian, C.N., Halachev, M., and Shiri, N. *Searching for Supermaximal Repeats in Large DNA Sequences.* Proc. of BIRD`08 - CCIS vol.13, Elloumi, M. et al. (Eds.), Springer Verlag, pp. 87-101, Austria, 2008

[**Manber and Myers, 1993**] Manber, U. and Myers, G. *Suffix Arrays: a new method for on-line string searches.* SIAM J. Computing, 22(5): 935-948, 1993

[**McCarthy and McDonald, 2003**] McCarthy, E., McDonald, J. *LTR_STRUC: A Novel Search and Identification Program for LTR Retrotransposons.* Bioinformatics 19(3): 362-367, 2003

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

[**Mehldau and Myers, 1993**] Mehldau, G. and Myers, G. *A system for Pattern Matching Applications on Biosequences.* Comp. Appl. in the Biosciences 9(3): 299-314, 1993

[**Morrison, 1968**] Morrison, D. *PATRICIA- Practical Algorithm To Retrieve Information Coded in Alphanumeric.* J. of the ACM, 15(4): 514-534, 1968

[**Myers, 1986**] Myers, E. *An O(ND) difference algorithm and its variations.* Algorithmica, 1(2): 251-266, 1986

[**Myers, 1996**] Myers, E. *Approximate Matching of Network Expressions with Spacers.* J. of Comput. Biol. 3(1): 33-51, 1996

[**Navarro, 1997**] Navarro, G. *Multiple approximate string matching by counting.* Proc. of WSP`97, pp. 125-139, Chile, 1997

[**Navarro, 1998**] Navarro, G. *Approximate Text Searching.* PhD Thesis, University of Chile, Santiago, 1998.

[**Navarro, 2001**] Navarro, G. *A Guided Tour to Approximate String Matching.* ACM Computing Surveys, 33(1): 31-88, 2001

[**Navarro and Baeza-Yates, 1998**] Navarro, G. and Baeza-Yates, R. *A practical q-gram index for text retrieval allowing errors.* CLEI Electronic Journal, 1(2), 1998

[**Navarro and Raffinot, 2003**] Navarro, G. and Raffinot, M. *Fast and Simple Character Classes and Bounded Gaps Pattern Matching, with Application to protein Searching.* J. of Comput. Biol.10(6): 903-923, 2003

[**Navarro et al., 2000**] Navarro, G., Sutinen, E., Tanninen, J., and Tarhio, J. *Indexing text with approximate q-grams.* Proc. of CPM`00 - LNCS 1848, Giancarlo, R. and Sankoff, D. (Eds.), Springer Verlag, pp. 350-364, Canada, 2000

[**NCBI, 2005**] http://www.ncbi.nlm.nih.gov/ Cited April, 2005

[**NCBI, 2007a**] http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/milestones.html

Cited April, 2007

[**NCBI, 2007b**] ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes. Cited April, 2007

[**NCBI, 2008**] NCBI's interface to BLAST. http://blast.ncbi.nlm.nih.gov/Blast.cgi

Cited April, 2008

[**NIH, 2000**] http://www.bisti.nih.gov/CompuBioDef.pdf Cited April, 2008

[**Ostell, 2005**] Ostell, J. *Databases of Discovery.* ACM Queue 3(3), April, 2005

[**Pearson and Lipman, 1988**] Pearson, W.R. and Lipman, D.J. *Improved Tools for Biological Sequence Analysis.* In Proc. of Natl. Acad. Sci., vol. 85, pp. 2444–2448, 1988

[**Phoophakdee and Zaki, 2007**] Phoophakdee, B. and Zaki, M.J. *Genome-scale disk-based suffix tree indexing.* Proc. of SIGMOD`07, pp.833-844, China, 2007

[**Policriti et al., 2004**] Policriti, A., Vitacolonna, N., Morgante, M., and Zuccolo, A. *Structured Motif Search.* Proc. of RECOMB`04, pp.133-139, USA, 2004

[**Ristov, 2003**] Ristov, S. *A Note on Indexing DNA and Protein Sequences.* Proc. of IS`03, pp. 121-126, Slovenia, 2003

[**SeqAn, 2008**] http://www.seqan.de Cited December, 2008

[**Sinha et al., 2008**] Sinha, R., Puglisi, S., Moffat, A., and Turpin, A. *Improving Suffix Array Locality for Fast Pattern Matching on Disk. Proc. 28th ACM SIGMOD Intl. Conf.*, pp. 661-671, 2008

[**SMaRTFinder, 2007**] http://bioinf.dimi.uniud.it/software/software/smartfinder Cited July, 2007

[**SMOTIF, 2007**] http://www.cs.rpi.edu/~zaki/software/sMotif Cited July, 2007

[**TDD, 2005**] http://www.eecs.umich.edu/tdd/ Cited April, 2006

[**Thamildurai, 2007**] Thamildurai, A. *Efficient and Scalable Indexing Techniques for Sequence Data Management.* Master's thesis, Concordia University, Montreal, Quebec, Canada, 2007

[**Tian et al., 2005**] Tian, Y., Tata, S., Hankins, R.A., and Patel, J. *Practical methods for constructing suffix trees.* VLDB Journal, 14(3): 281–299, 2005

[**Trace, 2007**] Trace Archive Database. http://trace.ensembl.org. Cited July, 2007

[**TrEMBL, 2007**] TrEMBL Database. http://www.ebi.ac.uk/trembl. Cited October, 2007

[**Ukkonen, 1985**] Ukkonen, E. *Finding approximate patterns in strings.* J. of Algorithms, 6(1): 132-137, 1985

[**Ukkonen, 1995**] Ukkonen, E. *On-line construction of suffix trees.* Algorithmica 14(3): 249-260, 1995

[**Vmatch, 2007**] http://www.vmatch.de Cited July, 2007

[Weiner, 1973] Weiner, P. *Linear pattern matching algorithms.* Proc. of 14th Annual 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, San Francisco, 1999

[Wu and Manber, 1992] Wu, S. and Manber, U. *Fast Text Searching Allowing Errors.* Comm. of the ACM, 35(10): 83-91, 1992.

[Wu et al., 1996] Wu, S., Manber, U., and Myers, E.W. *A Subquadratic Algorithm for Approximate Limited Expression Matching.* Algorithmica 15(1): 50-67, 1996

[Yang et al., 2004] Yang, J., Wang, W., and Yu, P. *BASS: Approximate Search on Large String Databases.* Proc. of SSDBM`04, pp.181-191, Greece, 2004

[Zaki, 2000] Zaki, M.J. *Sequence Mining in Categorical Domains: Incorporating Constraints.* Proc. of CIKM`00, pp. 422-429, USA 2000

[Zaki, 2001] Zaki, M.J. *SPADE: An Efficient Algorithm for Mining Frequent Sequences.* Machine Learning Journal 42(1/2): 1-31, 2001

[Zhang and Zaki, 2006] Zhang, Y. and Zaki, M.J. *SMOTIF: efficient structured pattern and profile motif search.* Algorithms for Molecular Biology 1:22, November 2006