Enhanced Suffix Trees

for Very Large DNA Sequences

Si Ai Fan

A Thesis

In the Department of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University

Montreal, Quebec, Canada

August 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Si Ai Fan

Entitled: Enhanced Suffix Trees for Very Large DNA Sequences

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Rajagopalan Jayakumar

_____ Examiner
Dr. Gregory Butler

_____ Examiner
Dr. Dhrubajyoti Goswami

_____ Supervisor
Dr. Nematollaah Shiri

Approved by_____
Chair of Department or Graduate Program Director

_____
Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Date _____

ABSTRACT

Enhanced Suffix Trees for Very Large DNA Sequences

Si Ai Fan

Recent advances in bio-technology have provided rapid accumulation of biological DNA sequence data. New techniques are required for fast, scalable, and versatile processing of such data.

Suffix tree (ST) is a data structure used for indexing genome data. This, however, comes with a price: it occupies a space that is about 10 times more than the input size. Existing disk-based ST index techniques either suffer from data skew problem, like TDD and HST, or are not space efficient for very large sequences, like TRELLIS and B2ST. We propose a new disk-based ST index, called Compact Binary Suffix Tree (CBST), together with a construction algorithm, which can support DNA sequences of size up to 256 terabyte. The results of our numerous experiments indicated that, compared to existing ST and suffix array techniques, CBST is superior in speed, space requirement, and scalability. It is the fastest among the disk-based techniques for very large sequences.

.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Nematollaah Shiri who gave me the chance to pursue a Master's degree. Without his continual support and encouragement, finishing my thesis would be impossible. I thank him for teaching me how to do research, for all valuable advices, and his tireless and patience in reviewing and correcting my progress reports and this thesis.

I would like to thank Concordia University for providing me a dynamic environment for my study and research. I had easily access to the computing facilities in the database research labs as well as convenient access to the Concordia library, when I needed, and where I needed, being on or off the campus. I would also like to thank all fellow graduate students in the database research labs for their technical helps and advices when needed.

I wish to thank my wife LingFang Dong from my heart. Her continuous support was the key factor to overcome difficulties I faced during of my studies. She also obtained her master's degree from Concordia.

Finally, I wish to express my love and gratitude to our beloved daughters, Avery and Brianna, to whom I dedicate this thesis

Table of Contents

List of Figures

List of Tables

List of Abbreviations

2PMMS	Two-Phase Multi-way Merge-Sort
B2ST	Big string, Big Suffix Tree
BST	Binary Suffix Tree
BWT	Burrows-Wheeler Transformation
CBST	Compact Binary Suffix Tree
DIGEST	Disk-Based Genomic Suffix Tree
EM	Exact Match
ESA	Enhanced Suffix Array
HGP	Human Genome Project
ISA	Inverse Suffix Array
LCP	Longest Common Prefix
LRS	Longest Repeating Substring
LT	Lookup Table
NCBI	National Center for Biotechnology Information
SA	Suffix Array
ST	Suffix Tree
STEM	Suffix Tree Exact Match
STTD64	Suffix Tree Top-Down 64 bits
TDD	Top Down Disk-based
TSQS	Ternary-Split Quick Sort
TRELLIS	An anagram of the bold letters in the phrase: **E**xternal **S**uffix **TR**ee with suffix **L**inks for **I**ndexing genome-sca**L**e sequences
WOTD	Write-Only Top-Down

Chapter

1 Introduction

1.1 DNA Sequences

Ever since 1965 the book “Atlas of a Protein Sequences and Structures” [Dayhoff, 1965] was published, molecular biology has witnessed tremendous growth. Recent advances in sequencing technology have allowed the rapid generation and collection of DNA. A huge amount of bio-sequences have been and are being generated in laboratories all over the world. The Human Genome Project (HGP) [HGP, 2011] is an international scientific research project started in 1989. The main goal of HGP is to identify and map approximately 20,000 to 25,000 genes of the human genome from both a physical and functional standpoint. The 1000 Genomes Project [1000 Genomes Project, 2011], launched in January 2008, is another international research project. Its objective is to sequence the genomes of at least one thousand anonymous participants from a number of different ethnic groups, As of late 2010, the project is in its production phase with a target of sequencing upwards of 2000 individuals. This will produce a huge collection of human genetic variations.

The entire DNA of an organism comprises that organism’s genome. As of Feb 15, 2010, National Center for Biotechnology Information (NCBI) published its GenBank release 182 through its web site. The current release contains 124 billion bases (1 base = 1 character) from 132 million sequence records [NCBI, 2011]. In addition, according to the GenBank release 162.0 (October 2007), the size of GenBank keeps

growing fast; the number of bases in GenBank has doubled approximately every 18 months. Figure 1 borrowed from [GenBank, 2011] clearly shows the exponential growth of the GenBank from 1982 to the 2009.

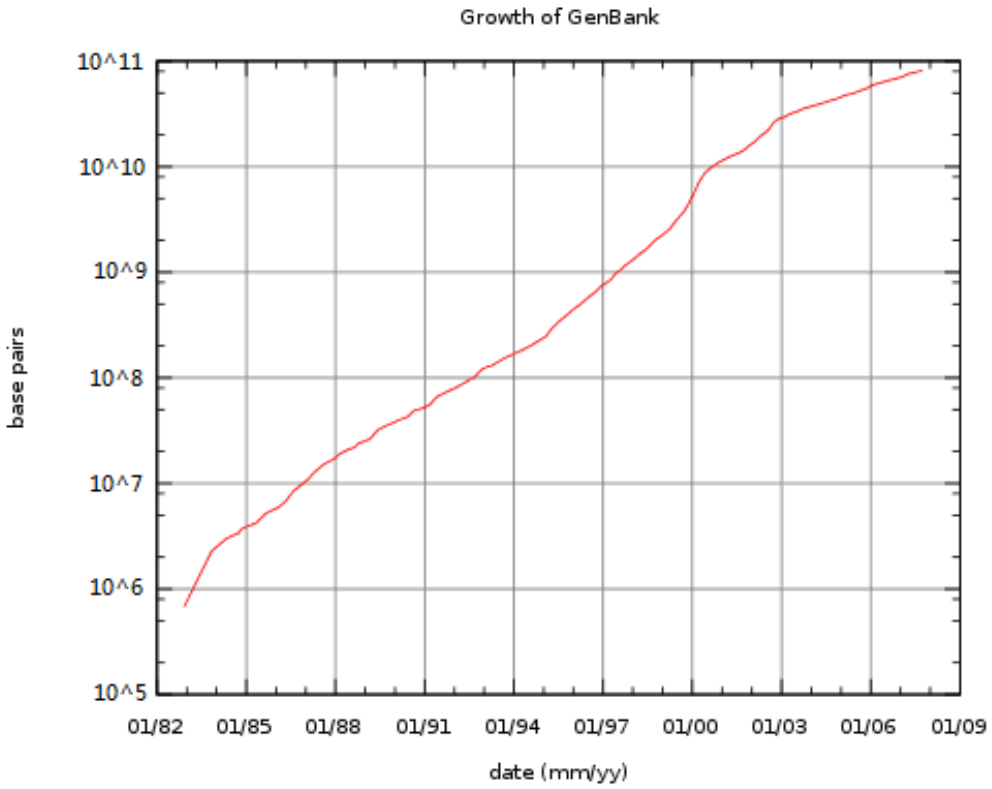


Figure 1. Number of Bases in GenBank Trend [GenBank, 2011]

How to analyze and understand this fast growing large data sets? As stated by Gusfield [Gusfield, 2004], "the shift to data-driven biology and the accumulation and exploitation of large-scale data has lead to the need for new computational technology (machines, software, algorithms, theory)." New techniques are required for fast, scalable, and versatile processing of biological sequences.

Although the DNA sequences are represented as strings of characters over the 4-letter alphabet {A, C, G, T}, they are fundamentally different from numerical sequences or

text data. Firstly, genomes sequence data does not have a structure (Comparing to natural languages), in that they cannot be meaningfully broken into parts/words. Thus, the traditional database and natural language processing technology are not applicable to genome sequence data. Secondly, the total size of inter-related information is several orders of magnitude larger in DNA than in typical natural language texts. For example, the long "volume" (chromosome) of a human genome is around 250 million characters. In addition, DNA sequences are a wide range in size, which is between several hundred nucleotides (e.g., expressed sequence tags, or ESTs) up to several billions (e.g., the entire human genome) [EST, 2011]. A real challenge is to develop efficient techniques to support various search tasks on short to very large sequences.

As genome sequences have no structure (Comparing to natural languages), it is essential in many string processing applications, to have an index on top of the raw sequences to speed up the process. Suffix tree (ST) and suffix array (SA) are the two most popular index techniques often employed in such applications. For biological databases, there are two main challenges. One is on constructing the index fast, and the other is on accessing the index efficiently. Our research aims to address these two problems.

Throughout this thesis, we use the term *memory-based* to refer to construction and search algorithms that require both the input sequence and its whole index to fit simultaneously in the main memory, and use *disk-based* to refer to algorithms that relax this requirement. We will use the terms string and sequence interchangeably in this thesis also.

1.2 Suffix Tree (ST) and Suffix Array (SA)

For indexing biological data, both ST and SA structures have attracted the attention of developers and researchers active in genomics and bioinformatics. Examples of techniques which use ST include 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], HST [Halachev et al., 2007], DIGEST [Barsky et al., 2008] and B2ST [Barsky et al., 2009], while examples of SA based techniques include ESA [Abouelhoda et al., 2004], Vmatch [Vmatch, 2011], DC3 [20], and LOF-SA [Sinha et al., 2008]. For an input string, a ST is a tree data structure which indexes and records all the suffixes of the string. A SA, on the other hand, is an array of integers indicating the starting positions of the suffixes of the input string in lexicographical order. Next we compare these two index structures and discuss the advantages of ST over SA.

The first advantage of ST over SA is its versatility for supporting many various search tasks, including exact match search, k-mismatch search, and k-difference search. For example, Apostolice [Apostolico and Galil, 1985] cites more than 40 references to ST, and Gusfield [Gusfield, 1997] discusses more than 20 applications of ST in the area of bioinformatics. Examples of these search tasks include looking for various types of repeats in a single sequence, finding the longest common prefix (LCP) subsequence of several sequences, and shortest superstring problem. Abouelhoda [Abouelhoda et al., 2004] proposed enhanced suffix arrays (ESA) to address the same search problems as an alternative to replace ST, criticized for its large index size. However ESA index includes several additional tables in addition to the basic SA table. It is also known that on average ESA requires 12N space, for an input string of size N for

storing all those tables.

Secondly, compared to SA index, ST index exhibits a larger capability for the index construction and better locality of reference when used by disk-based search operations. As the amount of biological data being generated is growing at phenomenal rates, maintaining an index in memory may no longer be feasible. We need to have an efficient external index algorithm. Examples of such disk-based algorithms include TDD [Tian et al., 2005], TRELLIS [Phoophakdee and Zaki, 2007], HST [Halachev et al., 2007], DIGEST [Barsky et al., 2008], and B2ST [Barsky et al., 2009], all of which are capable of building genome-scale level ST index in a 2GB RAM limitation. B2ST [Barsky et al., 2009] is a more recent technique that supports this genome-scale level sequences. For input sequences of size 6GB, B2ST was used to build the ST in 8 hours on a typical desktop computer with 2GB RAM. Vmatch is a powerful in-memory tool that implements the ESA algorithm, but its application is limited, in our experiments, to sequences up to 250MB on a computer with 2GB RAM. DC3 [Dementiev et al., 2005] extended SA construction algorithm to an external technique which can handle sequences up to 4GB, but its pipeline method is designed for multi-processors with multiple disks units. On a normal desktop, it has been shown that DC3 is inferior to TDD in construction time [Tian et al., 2005]. In addition, DC3 only generates the basic SA, which unlike ST solutions, cannot support many search operations. In addition, for longer sequences, query processing based on SA index exhibits poor locality of reference, leading to inefficient disk I/Os due to performing binary search in the array [Sinha et al., 2008].

While ST and SA are the two major index structures for biological sequence data, we can convert ST to SA and vice versa, under certain conditions, as shown in [Farach et

al., 2000]. For a ST, we can traverse the ST by a depth-first order and save the offset of a suffix inside a string to an array. This yields the SA of the string. On the other hand, assuming that the SA is augmented with the length of the LCP between consecutive suffixes in the SA, we can incrementally insert a suffix to a ST using the LCP length information. We will elaborate on this in the following chapters.

A suffix tree (ST) index not only records all distinct substrings of a given string, but also exposes the internal structure of the string in such a way that when exploited provides efficient solutions to versatile sequence analysis problems. These problems include the exact match (EM) search as well as various approximate search problems which are more complex than exact match.

1.3 Motivation

Since the initial proposal of Weiner [Giegerich and Kurtz, 1997] to use a suffix tree as an explicit index, data-driven biology and the accumulation and exploitation of large-scale data has resulted in much improvement in ST indexing techniques. For example, TDD [Tian et al., 2005], TRELLIS [Phoophakdee and Zaki, 2007], HST [Halachev et al., 2007], and DIGEST [Barsky et al., 2008] to B2ST [Barsky et al., 2009], are external ST index algorithms which can build ST for the entire human genome in a couple of hours [Barsky et al., 2009]. The approach of these external techniques is as follows. First, they divide a long sequence into smaller partitions, each of which are treated separately, given the amount of main memory available in a typical desktop computer. For each partition, they build a ST in isolation of other partitions. The STs so created are then merged to build the final ST index for the

whole string.

Until B2ST [Barsky et al., 2009], all the disk-based ST algorithms had either data skew problems, or were limited to handle up to genome-scale level sequences, mainly because of the partitioning and merging methods they used or/and the memory bottleneck problems. We will discuss details of these limitations and causes later. B2ST [Barsky et al., 2009] is a more recent ST algorithm, which is capable of handling very large strings using a typical personal computer. Using B2ST, it took only about 8 hours to build the ST for a 6GB input sequence [Barsky et al., 2009].

We studied several disk-based index representations and construction techniques and noted that we could improve both aspects. The TRELLIS index needs 36 bytes per symbol in the worst case, and can handle DNA strings of up to 4GB. The B2ST index has the same layout as DIGEST, and can support very long sequences, but its index size is about 48 bytes per symbol. Considering also the intermediate data produced by these algorithms, B2ST needs 1.3 terabytes to build the ST index for a 10GB sequence. Although the external disk space is much cheaper than main memory, this 1.3 TB data results in increased I/O cost, which in turn reduces the index construction and search operations time. The design objectives of the HST index were reducing the intermediate space required during the index construction as well as provision for fast search operations. For this, the authors studied the disk access patterns during the index construction and search [Halachev et al., 2005]. For DNA sequences, HST requires 13N bytes per character on average, for input of size N. Besides, HST uses a "double" node index structure and saves the suffix depth information to its leaf nodes to speed up the search algorithms. By design, HST shows good locality of references for the disk-based indexes and also exhibits good performance during query

operations.

As our research goal is to develop a ST index technique that is efficient and also capable of handling very long sequences on a typical desktop computer, to meet the data-driven biological database requirements. At the same time, the ST index needs to support efficient query operations. The question is: Can we design a new ST index representation that is space-saving and efficient for query operations, like HST [Halachev et al., 2007]? And at the same time, can we have an efficient and scalable index construction algorithm, like B2ST [Barsky et al., 2009]? This thesis attempts to answer these questions.

1.4 Research Contributions

As mentioned, for data-driven biological datasets, the real challenge is to develop suitable techniques to support efficient search in short to very long sequences. The main contribution of this thesis is development of a compact binary ST indexing technique, which is efficient and scalable for short to very long sequences. More details are as follows: We study and analyze the current ST index techniques, their design and implementations, and propose a ST tree index representation based on the binary alphabet, which we call as the compact binary suffix tree (CBST). It requires 9 bytes per node and 18 bytes per symbol for an input, and supports sequences as large as 256 TB. We introduce our CBST index in Chapter 3.

We develop a ST index construction algorithm for CBST index representation, called CBST algorithm. The algorithm is an extension of B2ST [Barsky et al., 2009]. We show the effectiveness of CBST technique for in-memory and on-disk data. Our

CBST algorithm is efficient on short sequences (like single chromosome-scale sequences) and on very long sequences (like the entire human genome and above). We introduce it in Chapter 3.

Besides, we also consider the two suffix sorting algorithms, namely Msufsort [Michael and Simon, 2008] and Qsufsort [Larsson and Sdakane, 1999], and explain why our CBST algorithm uses Msufsort for sorting suffixes, and not Qsufsort which was used by B2ST algorithm. We explain the reasons in Chapter 3, Msufsort is faster and more space-efficient than Qsufsort.

We also implement an exact matching (EM) searching algorithm that uses the CBST index. The EM algorithm we developed is similar to the one proposed in [Halachev et al., 2005], which extends the memory-based ST exact match (STEM) algorithm to disk-based and uses efficient buffering strategy. Experiments performed to evaluate the CBST search algorithm indicate increased performance, and good locality of references to the disk during query operations for the CBST index.

The results of our extensive experiments and their analysis in this research to study the proposed index representation together with the construction and search techniques developed indicate that our CBST algorithm is a desired efficient and scalable disk-based ST index. It is the fastest disk-based ST index construction algorithm so far.

1.5 Outline of the Thesis

The organization of the thesis is as follows.

In Chapter 2, we provide the necessary background, review related work, and discuss a conventional suffix tree and its corresponding binary suffix tree (BST), and the features of a BST. In addition, we study several disk-based suffix tree techniques and their corresponding index representations. We compare them with their advantages and disadvantages.

In Chapter 3, we propose our compact binary suffix tree (CBST) index representation. We will also discuss the advantages and disadvantages of CBST in comparison with existing ST based representations. At the same time, we provide technical details of the proposed CBST model together with construction and search algorithms. We also present an exact match search algorithm based on the CBST index.

To evaluate the performance of the proposed technique, we conducted numerous experiments using short to very long sequences. Chapter 4 describes the experiment setup, the data and queries used in our experiments. We present the results of these experiments and compare them with the best-known alternatives.

Chapter 5 includes a summary of our contributions, concluding remarks, and an outline of possible future work.

Chapter

2 Background and Related Work

In this chapter, we recall some definitions and concepts regarding the suffix tree (ST) index. This includes a formal definition of conventional suffix trees for sequences based on some particular alphabets. We then define *binary ST* (BST) that is based on binary alphabet, which forms a basis for the development in this thesis. Besides, we also review major disk-based ST algorithms and their corresponding ST index representations that are related to our work.

2.1 Conventional Suffix Trees

A string S is a sequence of N symbols over an alphabet Σ. We use the symbol $ (not included in Σ) as the terminal character, used to mark the end of S.

Definition 1: Given a sequence S of size N, a suffix S_i of S is the substring S[i, N] of S that begins at position i, where $0 \leq i \leq N$. Thus S_0 = S and S_N = $. Each suffix can be uniquely identified by its starting position. For example, for S = ACGTG$, the suffix S_1 will be CGTG$, S_2 will be GTG$, S_3 will be TG$, etc.

Definition 2: A prefix P_i of a suffix S_x is the sub-string [0, i] of S_x, where 'i' is less than the length of S_x. For example, for S = ACGTG$, the prefix P_3 of the suffix S_1 (=CGTG$) will be CGTG, and P_1 of the suffix S_2 will be CG, etc.

Definition 3: The longest common prefix (LCP) of two suffixes S_i and S_j is a substring $S_{[i, i+k]}$ such that $S_{[i, i+k]} = S_{[j, j+k]}$, and $S_{[i, i+k+1]} \neq S_{[j, j+k+1]}$. Thus, the prefix $S_{[i, i+k]}$ (or $S_{[j, j+k]}$) is the LCP of the suffixes S_i and S_j. And their LCP length is 'k'. In another words, the LCP of two suffixes is the longest prefix that is shared by this two suffixes. We denote the LCP of suffix S_i and S_j as LCP (i,j), and the LCP length as |LCP (i,j)|. For example, for S = ACGTG$, considering S_2 = GTG$, S_4 = G$, we have that LCP(2,4) = G, and its length |LCP(1,4)| = 1.

Definition 4: A suffix array (SA) of a sequence S is an array that stores the offsets of all the suffixes of S in lexicographical order. If each offset of a suffix of S is represented by an integer, then the SA will be an array of integers. Thus, a SA of S holds all the integers 'i' in the range [0, N], where 'i' represents S_i. Note that the suffixes themselves are not stored in this array but are rather represented by their start positions in S. For example, for S = ACGTG$, the SA = [0, 1, 4, 2, 3]. The LCP between each consecutive SA pairs can be kept for building suffix tree. For example, LCP (0,1)=0, LCP(1,4)=0, LCP(4,2)=1, etc.

Definition 5: A suffix tree (ST) (also called PAT tree or, in an earlier form, position tree) is a data structure that presents the suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations [Wikipedia, 2011]. The ST of the sequence S is an edge-labeled tree with N leaves (or suffixes of S). Each edge records the start and end positions in S (which represents a substring of S). Each internal node in the ST represents an end of the LCP for its children leaf nodes (or suffixes of S). The

path from the root to a leaf defines a suffix of the sequence S.

Figure 2 shows the suffix tree for the sequence S = ACGTG$. In the figure, the label on an edge lists the substring to the internal nodes or leaves. By traversing the whole tree, going from the root down the internal node 1, we can get the LCP 'G' for the two suffixes S_4 and S_2, from the root down to a leaf, like S_4, we can get a suffix S_4=G$ of S. From the tree, we have the same number of leaves as the number of suffixes of S and they are in the lexicographical order from the left to right. Thus, if we traverse the whole tree from the root to all the leaves using a depth-first approach, , we get S_0, S_1, S_4, S_2, S_3, and record them in a SA of S. This yields [0,1,4,2,3].

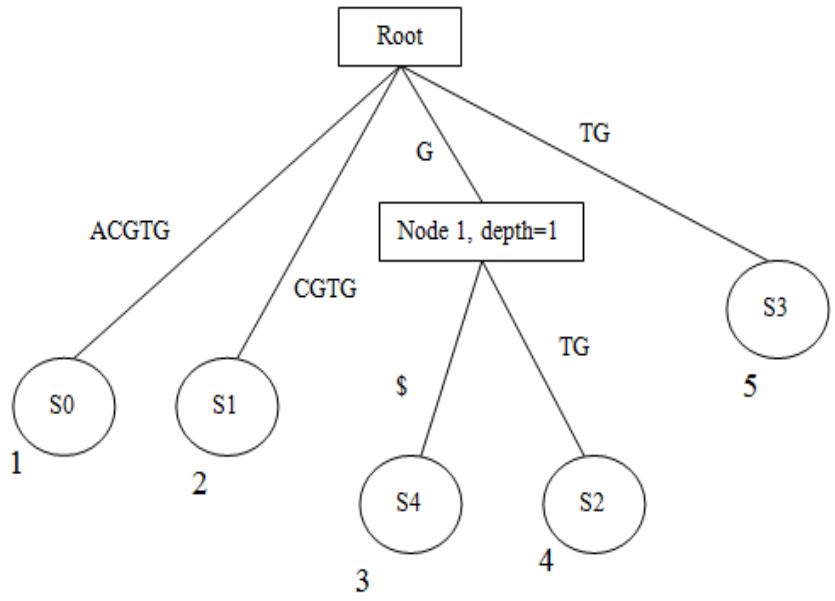


Figure 2. The suffix tree for the sequence S = ACGTG$

In this chapter, we will also study different representations of suffix trees in memory and on disk. Below we recall some definitions related to suffix trees.

Definition 6: For a node v in a ST for a sequence S, the leaf set of v, denoted L(v), contains the positions of all its leaves and its children leaves in sequence S at which we can find the suffix denoted by the edge labels from the root of ST to node v.

For example, for the leaf node 4 in Figure 1, we have that L(4)={2}. For branch node 1, we have L(branch node 1)={2,4}.

Definition 7: The depth of a node v, denoted by depth (v), is defined as the length of the path, which in turn is defined as in number of characters on the labels of the edges from the root to the node v. For example, depth (4) = 3 and depth (Node 1) =1.

Definition 8: For any node v in a ST, the left pointer of v, denoted by LP(v), is defined as min L(v) + depth (the parent of v). For example, LP(1) = min L(1) + depth(Root) = 2+0 = 2.

2.2 Binary Suffix Tree (BST)

Any alphabet Σ can be represented in binary by representing each character as a binary sequence of b = log|Σ| bits. This can be done in linear time [Farach and Muthukrishnan, 1996]. This means any string can also be represented by a binary string. For example, we only need 2 bits to represent the symbols in the alphabet set {A, C, G, T} of DNA sequences. In our work, we use the following encoding rule:

A = 00, C = 01, G = 10, T = 11.

If we build a suffix tree using binary strings, since we only have 0 and 1 in the binary

alphabet, any internal node in the tree may only have two children. Thus, we get a binary ST (or BST, for short). (We remark that the suffix binary search tree mentioned in [Irving and Love, 2003] is different from our BST, and stands for balanced search tree.) Figure 3 lists the corresponding BST for the string S = ACGTG$. We can see that this tree has the same collection of leaves as the tree in Figure 2. This is true because we build the BST with the same number of suffixes for the same string S.

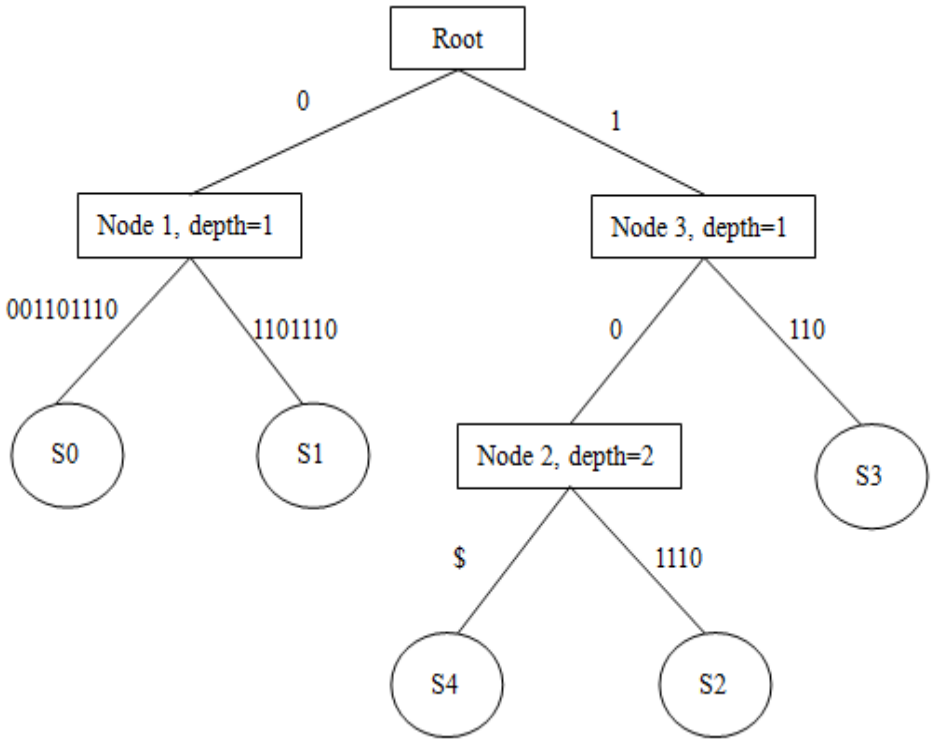


Figure 3. The BST for S = ACGTG$

Compared to conventional suffix tree, BST has the following features:

Each internal node can have maximum 2 children nodes. This allows the final tree to be organized in arrays, which in turn supports fast tree traversals, since the corresponding child can be located in constant time. In Chapter 3, we will show how

to build a BST.

In a BST for a sequence S, the total number of leaves and branch nodes is linear in the length of S. If S has N suffixes, the BST of S will have N leaf nodes and N-1 branch nodes (including the root). In addition, the total number of leaf and branch nodes is independent of the alphabet of the sequence. For example, if a DNA sequence and a protein sequence (with an alphabet of size 23) have the same length N, their corresponding BST will have the same N leaves and N-1 branch nodes. Given a certain representation of a branch and leaf node, the size of a BST index is linear in the number of the characters in the sequence, and is independent of the alphabet. We will present our proposed BST index structure in Chapter 3.

If all suffixes are sorted, BST can be constructed incrementally by following the path from the root to the last added suffix. As it will be made clear in Chapter 3, we can locate the insert point along the path by the LCP length information, and add a suffix to the right child of the insert point.

This BST representation can support many common string queries. For example, in order to find occurrences of a pattern in string S, we rename the pattern to a sequence of bits by using the same policy applied to S while building the BST, and match these bits along the path starting at the root. Once we reach an internal node, all the leaves of this node are the answers. Also, if we are looking for the longest repeating substring (LRS) of S, and the alphabet contains characters, each represented by “b” bits, we find the internal node of the greatest depth, say “d”, from the root. We then calculate the LRS (with respect to the original alphabet) as LRS = d/b.

2.3 Suffix Tree (ST) Indexing Techniques

Suffix trees are data structures used which index all the suffixes of a given sequence. It is a versatile structure that can be used to evaluate a wide variety of queries on sequence datasets, including evaluating exact and approximate match search operations, and finding repeat patterns. However, methods for constructing suffix trees are often very time consuming, especially for those large suffix trees that do not fit in the available main memory.

Suffix trees, originally called position trees, were introduced by Weiner [Weiner, 1973]. Shortly after, a more space efficient algorithm was proposed by McCreight [McCreight, 1976]. Later on. Ukkonen proposed a variant of McCreight's algorithm which was much easier to implement [Ukkonen, 1995]. In [Giegerich and Kurtz, 1997] it was shown that these three proposals are similar in algorithmic ideas. The algorithms are linear in construction time and memory based, i.e., they require both the sequence and the ST index to fit and reside in the main memory. A key point to have an implementation of the index construction algorithm that runs in linear time, is the requirement to use suffix links [Gusfield, 1997]. However, it has been shown in [Hunt et al., 2002] that the presence of suffix links results in reduced performance due to random accesses to the tree during the index construction. Once some of these variants data structures outgrow the main memory by accessing the data on disk, the access time to disk-based arrays vary significantly depending on the relative location of the data on disk, and the total number of random disk accessed is O(N), which is extremely inefficient. This results in poor performance and lack of scalability of the above three memory based algorithms for long sequences.

To address the above problem, several disk based ST index construction techniques

were proposed, including TDD [Tian et al., 2005], TRELLIS [Phoophakdee and Zaki, 2007], HST [Halachev et al., 2007], DIGEST [Barsky et al., 2008], and B2ST [Barsky et al., 2009]. These techniques do not consider or record suffix links during the suffix tree construction. (TRELLIS can recover suffix links after the ST is built). Amazingly enough, although all these disk based algorithms run in $O(N^2)$ in the worst case, they are much faster than linear time algorithms in practice, due to better locality of tree accesses.

To our knowledge, Hunt et al. [Hunt et al., 2002] was the first practical external ST construction algorithm, which is an incremental method that trades an ideal O(NlogN) performance for locality of access to the tree during its construction. The algorithm abandons the suffix links and partitioning the long sequence to shorter ones, for which the ST can be built in main memory. The output tree is in fact represented as a forest of several suffix trees. The suffixes in each such tree share a common prefix. Each tree is built independently and requires scanning of the entire input string for each prefix. This works well for non-skewed input data but fails if for a particular prefix length, the number of suffixes in a partition is significantly larger. This is often the case in DNA sequences with a large amount of repetitive substrings. For each possible prefix length, in order to keep the balance of each partition and allow the tree built under it to fit into the main memory, we can increase the length of the prefix. This exponentially increases the total number of partitions, which in turn increases the total number of input string scans.

TDD [Tian et al., 2005] suggests a Top-Down, Disk based ST construction algorithm. This algorithm performances very well for the chromosome-level DNA sequences. The algorithm extends the Write-Only Top-Down (WOTD) [Giegerich, et al., 2003]

index representation and incorporates the fixed-length prefix partitioning method described in [Hunt et al., 2002]. To overcome the 1 gigabyte limit sequences, TDD introduces two additional bitmap arrays to represent the rightmost bit and the leaf bit, and the remaning 32 bits for the tree node representation. Thus TDD can handle long sequence to 4 gigabyte in theory. We will study details of the TDD index representation in the next chapter. TDD manages more efficiently the memory buffers and is a cache-conscious method which performs very well for many practical inputs. TDD has shown to be superior to other ST and SA solutions like [Bedathur and Haritsa, 2004] and [Dementiev et al., 2005]. It is reported, for the first time, the suffix tree for the entire human genome was constructed in about 30 hours using a typical desktop [Tian et al., 2005].

HST [Halachev et al., 2007] further extends TDD index representation by considering a two level index structure and employing a dynamic buffering strategy, that resulted in improved index construction and search performance. While TDD uses two bitmap arrays to overcome the 1GB index limitation, HST embodies the two bits to its 64 bits suffix tree node representation, either a branch node or a leaf. This also allows HST to handle sequences of up to 4GB. HST improves the on-disk STs, called STTD64 [Halachev et al., 2007]), proposed by the same authors by construction a second index on the STs (called the lookup table) to speedup search operations. We will have a closer look at HST in Chapter 4.

While TDD [Tian et al., 2005] and HST [Halachev et al., 2007] are scalable to long genome-scale level sequences, they perform considerable random disk accesses to the input string during the tree construction. To reduce the negative impact of this on performance, both techniques required the input sequence to reside in the main

memory for better references. This limits the input string size to that of the main memory. This also explains their use of compression methods for long sequences to keep the sequence in memory by encoding the alphabet to binary format. Another problem is their partition size and the on-disk tree layout due to the fixed-length prefix technique. A long prefix may result in increased the number of partitions, with possibly many smaller size, and hence poor space utilization. Also a short prefix may cause some partitions to be larger than the memory. This requires buffering the index nodes and hence incurring increased additional disk I/O cost. In addition, different partitions have different index sizes, some possibly significantly larger than the main memory. This poses some problems when loading the sub-tree into main memory for querying. If a sub-tree cannot be loaded into the main memory, the depth first traversal of such trees requires multiple random accesses to different levels of index nodes in the disk, and hence poor performance.

TRELLIS [Phoophakdee and Zaki, 2007] was proposed to address the above problems. It adopts an innovative method to partition the input to avoid data skew problem by using a variable-length prefix method. It first computes all the variable-length prefixes by scanning the input sequence multiple times. During each scan, the prefixes up to a certain length are saved, such that a partition and the ST built afterwards can be processed entirely in the main memory. It adopts Ukkonen's algorithms [Ukkonen, 1995] for creating the sub-trees. Once an independent suffix tree for each partition is built in memory, it writes to disk the different sub-trees correspond to the different variable-length prefixes. The subtrees of all the partitions are then merged into a shared prefix subtree for the entire input string. TRELLIS also contains a post-processing step for recovering all the suffix links. TRELLIS has been shown to be superior to TDD [Phoophakdee and Zaki, 2007]. On the same

computer, TRELLIS completed the construction of the index for the entire human genome in about 4 hours, and additional 2 hours for recovering all the suffix links.

TRELLIS also introduces another important technique during its partitioning phase, as follows. In order to guarantee that the ST of a partition includes explicitly all the suffixes from the partition, instead of stopping the tree construction for a partition when exactly all the characters inside the partition have been read, the algorithm continues to read some of the characters from the next partition, until enough suffixes are explicitly obtained. That is, while building the ST for a partition, TRELLIS continues to read characters from the next partition until it encounters a unique prefix. This interesting idea was adopted in DIGEST [Barsky et al., 2008] and B2ST [Barsky et al., 2009], discussed in Chapter 4.

TRELLIS is capable of handling sequences of up to 4 GB only. This limitation is due to its index representation which are suited for sequences no longer than 4GB. Another limitation of TRELLIS is that it requires the input sequence to be in the main memory for better references. As mentioned, in order to construct the entire human genome (about 3 GB) using a computer with 2GB RAM, TDD [Tian et al., 2005], HST, and TRELLIS compress the input sequence to encode the alphabet symbols {A,C,G,T} in binary, using 2 bits. Thus the entire human genome needs 725 MB in the main memory.

To overcome this memory bottleneck, two other algorithms TRELLIS+ [Phoophakdee and Zaki, 2008] and DIGEST [Barsky et al., 2008] have been proposed recently which use a buffering strategy. TRELLIS+ buffers some parts of the sequences that probably need to be accessed by the merging procedure and some initial characters of each leaf node. On the other hand, DIGEST buffers a predefined fixed-length prefix

of each suffix. It was noted [Barsky et al., 2008] that references to the input sequence happen only when comparing two suffixes to get their LCP in order to locate the merging point in current being built ST. The buffered prefix of a suffix could help get the LCP length of current suffix and the one added just before it. Thus, both algorithms relax the requirement that the whole input sequence has to be in the main memory. Using buffering, TRELLIS+ limits its access to the on-disk input sequence to 5%, while DIGEST limits this access further to 2%. It has been shown that DIGEST outperforms TRELLIS+ by about 40%. We remark that for a 10GB sequence, the 2% disk accesses translates to 500 million random disk I/Os. This significantly degrades the performance of the algorithms. Thus, both TRELLIS+ and DIGEST are limited in practice to handle very long sequences. The technique proposed in DIGEST to incrementally build the BSTs, for a given sequence, was extended and used in B2ST [Barsky et al., 2009]..

The B2ST was more recently proposed for handling very large sequences under limited resources. It divides an input sequence to equal chunks and builds suffix arrays (SA) for each chunk. At the same time, B2ST also collects the LCP length and suffix order information by sorting all possible chunk pairs. During the final merge phase, B2ST obtains the lexicographical global order for all the suffixes. This is done without needing to refer to the input any more. It was reported that B2ST was able to build the ST index for a 12GB input sequence on a typical desktop in just 25 hours [Barsky et al., 2009]. To the best of our knowledge, B2ST is the fastest disk-based ST algorithm to which we compare our work. This motivated our work in this thesis to develop a faster and scalable ST index construction technique for long sequences.

2.4 Suffix Tree Index Representations

We review several disk-based ST index representations, and discuss their advantages and disadvantages. In the next chapter, we will then present the compact binary suffix tree (CBST) index representation, which we proposed in this research and takes advantages of the current disk-based ST representations.

ST Storage Requirements

Given a sequence S of size N on an alphabet set Σ, its ST will have exactly N leaf nodes and at most N-1 branch nodes. Thus, the maximum number of nodes is linear in N. It is common to represent the node in a ST together with the information about an incoming edge label. Each node, therefore, contains two integers representing the start and end positions of the corresponding substring of S. In fact, it is enough to store only the start position of this substring as the end position can be deduced from the start position of the child (for a branch node) or is simply a suffix offset in S (for a leaf node). In a straightforward implementation, each ST node has pointers to all its child nodes. These pointers can be represented as an array, as a linked list, or as a hash table [Gusfield, 1997].

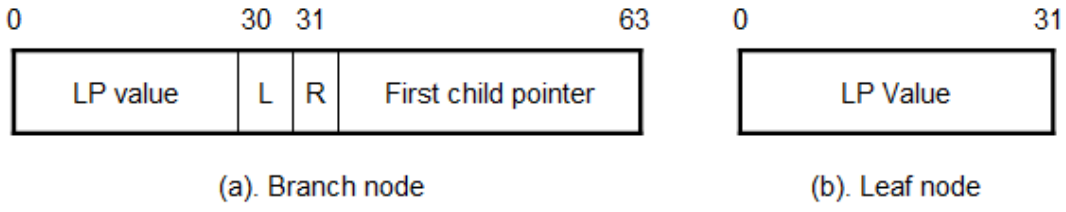
Compared to linked-list index representation, hash tables and arrays are more efficient data structures for tree traversals, since the corresponding child node can be located in a constant time. While it is easier to implement a ST index as an array, optimization is required for the representation, since otherwise, each node can have |Σ| entries

pointing to its children, plus one entry to represent the start position of the edge-label substring. If we use an integer to represent an entry, since there are at most 2N-1 nodes in the ST for sequence S, the total storage space required is (2N-1)(|Σ| +1) integers. For a DNA sequence, which includes four symbols (|Σ| = 4), this requires 5*(2N-1) integers, which is 20(2N-1) bytes of storage for the input of N bytes.

In this chapter, we review major array based ST index representations. They include WOTD [Giegerich, et al., 2003], TDD [Tian et al., 2005], HST [Halachev et al., 2007], TRELLIS [Phoophakdee and Zaki, 2007], and B2ST [Barsky et al., 2009]. We will then propose our CBST index representation.

WOTD Index Representation

The WOTD ST representation [Giegerich, et al., 2003] derives its name from a ST construction approach called *Write Only Top Down*. It is implemented as a linear array of 32-bit elements. As shown in Figure 4(a), each branch node in the ST occupies two adjacent elements, while a leaf node is represented as a single array element, shown in Figure 4(b).



(L means leaf bit and R means rightmost bit)

Figure 4. WOTD Index Representation

For a branch node, the first 32-bit element stores in 30 bits the left pointer value, defined in Chapter 2, and the 2 bits, called *the leaf bit* and *the rightmost bit*. A leaf bit value 1 indicates the node is a leaf; otherwise it is a branch node. A rightmost bit value 1 indicates that this ST node is the rightmost child of its parent. The second 32-bit element of a branch node stores a pointer to its first child, which points to its first child position in the WOTD index.

For a leaf node, the 32-bit element stores the same information as stored in the first element allocated for a branch node: the left pointer value, the leaf bit (always set to 1), and the rightmost bit.

As the ST nodes are evaluated and stored in a top-down, left to right manner, the advantage of the WOTD index representation is that the edge labels can be found in constant time, using the left pointer values. The WTOD ST representations is the most space efficient index [Giegerich, et al., 2003]. In the worst case, it only needs 12 bytes per character for storage space. It was shown that for real-life DNA sequences, this index requires about 9 bytes per character on average.

A disadvantage of the WOTD index representation is its limitation to handle up to 1 gigabyte long sequence theoretically due to its 20 bits for storing the left pointer. It is suitable only for a memory based construction algorithm. We next introduce TDD, an external ST index which extend this representation to disk based, and overcomes this limitation.

TDD Index Representation

Figure 5 shows the TDD representation. It overcomes the 1 GB limit of the WOTD representation by introducing two additional bitmap arrays: the rightmost bit and the leaf bit, recorded for each ST node. Compared with the WOTD index, all the 32 bits of the first element of a branch node and a leaf node are available for recording the left pointer value. This makes TDD capable of handling sequences of size up to 2^{32} characters, i.e., 4 gigabyte.

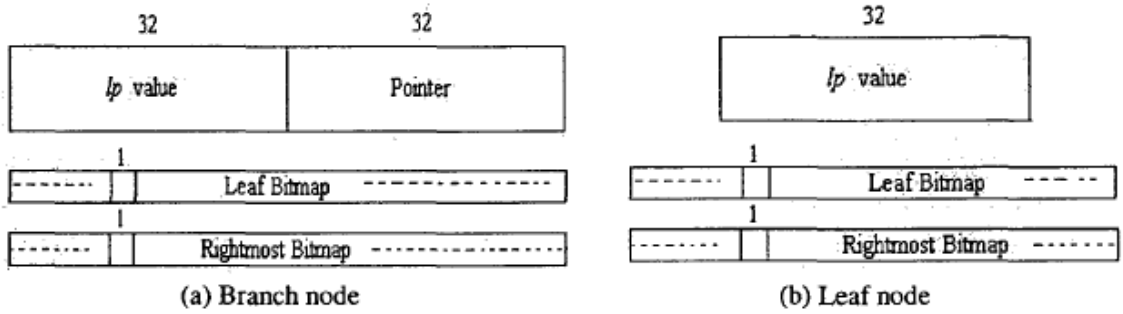


Figure 5. The TDD Index Representation [Halachev., 2009]

Although the TDD index representation extends the WOTD index to a disk-based algorithm, it is inadequate to support efficient disk-based query operations. We introduce the TRELLIS and the HST ST index representations next, which improve the search performance.

TRELLIS Index Representation

As the TDD, the TRELLIS representation [Phoophakdee and Zaki, 2007] allows for handling sequences of size up to 4GB in theory. As introduced in Chapter 2,

TRELLIS ST construction algorithm partitions an input sequence by a variable-length prefix of suffixes. It merges all the partition STs to form a final forest of STs that shares the same prefix. Further, each prefixed ST consists of two files, one for recording the branch nodes, and the other for recording the leaf nodes. Figures 6 (a) and (b) show the structures of branch and leaf nodes in TRELLIS.

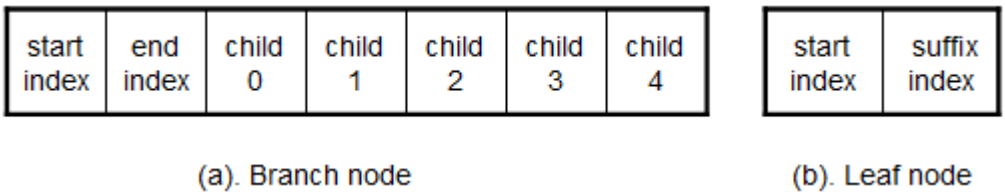


Figure 6. TRELLIS Index Representation

Each branch node occupies seven 4 byte elements, for a total of 28 bytes. The first two elements represent an edge [start index, end index] between a branch node and its parents. The next 5 elements are allocated for the branch node, representing its outgoing edges. The child0, child1, child2, child3, and child4 denote the child with edge starting with $, A, C, G, T characters (corresponding to DNA sequences) respectively. The child can take either one of the three possible values:

(a) 0 or NULL, indicating no child.

(b) A number in the range [1, t], denoting a leaf node, where ‘t’ is a threshold obtained during index construction.

(c) A number larger than t, denoting an internal node.

Each leaf node occupies two 4 byte elements, for a total of 8 bytes. It represents the edge [start index, end of input string] between a leaf and its parents.

Compared to the TDD index representation, TRELLIS avoids the data skew problem due to the variable-length prefix partitioning technique. It stores all the child elements in a branch node which yields more efficient disk-based traversal of the ST. As a result, the exact match search using TRELLIS is faster compared to the memory-based TDD search [Phoophakdee and Zaki, 2007]. TRELLIS also provides an extra option to recover its suffix links after the ST has been built. Suffix links help to speed up the ST traversal in some search problems.

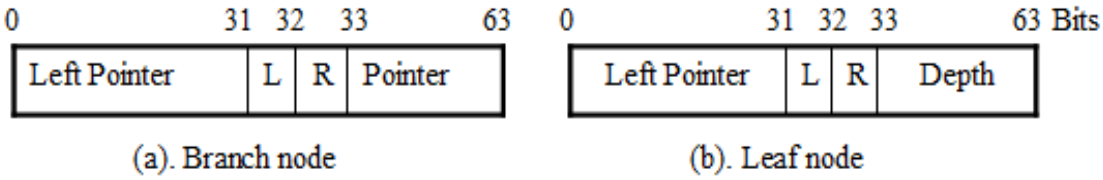
A disadvantage of TRELLIS is its index size being large and being proportionate to the alphabet size. Even for the small, five symbol DNA alphabet (which includes A, C, G, T, $), the size of the TRELLIS index for real-life sequences is on average 25N bytes (and up to 50N bytes, if the suffix links are recorded as well), where N is the number of characters in the input sequence. Being proportion to alphabet size, the TRELLIS index is thus more suited for DNA sequences.

HST Index Representation

HST [Halachev et al., 2009] index representation combines a lookup table (LT) and the suffix tree index STTD64, that was proposed earlier [Halachev et al. 2007]. The LT serves as an index to the STTD64 index.

STTD64 is an extension of TDD which outperforms TDD by integrating the leaf and rightmost bits to the node representation. Each STTD64 node is represented as a single 64-bit record, regardless of being a branch or a leaf node. Figure 7 shows the

structures of branch and leaf nodes in STTD64. For both types of nodes, the first 32 bits are allocated for storing the left pointer. 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, while a leaf node stores its depth information. Thus, the STTD64 representation requires 16 bytes for a suffix in the worst case. On average, however, it requires 13.5N bytes for a DNA sequence of size N [Halachev et al., 2007].



('L' means leaf bit, 'R' means rightmost bit)

Figure 7. STTD64 Index Representation

The LT index of HST is implemented as an array of pointers to STTD64 nodes. It is used as a reference to the disk-resident STTD64 to avoid extra disk I/Os during query processing.

The HST index has the same capability as TDD and TRELLIS and can support up to genome level sequence, e.g. 4 gigabyte. HST supports efficient query processing by saving the depth information in the leaf nodes and having good locality of references with the LT index. As a result, the exact match and k-mismatch search tasks on HST are faster than these operations with TDD and TRELLIS indexes [Halachev et al., 2007].

As is the case with TDD, a disadvantage of HST is the data skew problem associated

with this index due to its fixed-length prefix partition technique. Secondly, its LT index is constructed after the STTD64 index is built and stored to the disk; the ST needs to be read into memory again and perform a partial exact match search. Thus, it takes another round of disk I/Os, which results in slower index construction algorithm compared to TRELLIS. Thirdly, the HST index construction algorithm does not scale to input sequences larger than 4GB. The same limitation exists with TDD and TRELLIS. The B2ST index representation [Barsky et al., 2009] (same as DIGEST [Barsky et al., 2008]) studied next overcomes this limitation.

DIGEST and B2ST Index Representation

As DIGEST [Barsky et al., 2008] and B2ST [Barsky et al., 2009] share the same index layout, in the sequel we only consider B2ST. As in other disk-based indexes, B2ST organizes its ST tree nodes in an array data structure in both memory and the disk. However, B2ST is based on the binary suffix tree (BST). Figure 8 shows the structure of the nodes in the B2ST index.

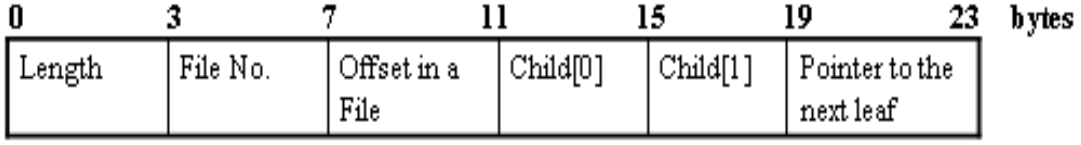


Figure 8. B2ST Branch and Leaf Node Representation

For both branch and leaf nodes, B2ST index representation includes 6 parts, each with

4 bytes, for total of 24 bytes. The first part is used to store the total number of symbols from the root to a branch node (Or the length, as all the symbols along the path from the root to the branch node defines a prefix of all its children nodes). The second and the third parts are used to represent a suffix of a sequence, defined as [File No., Offset in a File], i.e., [Partition No., Offset in a partition]. The next two parts represent two pointers for storing the left and right children location in the array. As B2ST index is based on BST, each ST branch node can only have two children nodes. The last part of the index stores a pointer for storing the next leaf location that has the same LCP length information as the current node.

Since [File No., Offset in a File] defines the total length of a sequence that B2ST can support, we can see that B2ST can support sequences up to 2^{64} in length, which is much larger than handled by any of the old ST index representations.

A disadvantage of B2ST index representation is that it is not compact. Its index size is 48 times the input size, in the worst case. Our experiments show that B2ST requires 45 bytes per character. We noted that a leaf node in B2ST does not have any children, a branch node does need to save its offset in the string neither, and the 6^{th} part that stores the next leaf location in B2ST index is not really necessary.

Considering the above disadvantages of B2ST, we propose a compact binary suffix tree (CBST) index representation, introduced next chapter, which also takes advantage of and deploys a number of techniques used in the development of the above disk-based index representations.

2.5 Summary

In this chapter, we recalled some definitions and concepts regarding the suffix tree (ST) index. We also introduced the *binary ST* (BST) that is based on binary alphabet, which forms a basis for the development in this thesis. Besides, we also reviewed major disk-based ST algorithms and their corresponding ST index representations that are related to our work.

we studied major ST index representations. WOTD is one of the most space-saving representations, however, it is only suitable for a memory based ST construction algorithm. TDD and HST indexes extend WOTD to disk based algorithms, both of which are capable to handle sequences of up to 4GB. However, both indexes suffer from data skew problem due to their fixed-length prefix partition technique. TRELLIS is as powerful as TDD and HST, but it is limited to DNA sequences since its index size is larger than TDD and HST, however it can grow even larger when the alphabet becomes larger as is the case for proteins. While B2ST (and DIGEST as well) is based on BST, it can support longer sequences than others could handle. However, the B2ST index representation is not compact. Our proposed compact binary ST (CBST) index representation improves this restriction. We introduce our CBST in the next chapter.

We summarize the advantages of the above disk-based ST index as below:

All these techniques store their STs files on disk in array format. As described, array based representations are more efficient for tree traversals, since locating a child node, which is done frequently during query processing, could be done in constant time.

Adopting a two level index structure can support efficient disk based query operations, for providing good locality of references to STs on disk, by saving disk

I/Os.

Saving valuable information in the ST nodes results in increased performance. As in HST, by storing the depth information in the leaf nodes, it can avoid extra jumps in traversals of the STs.

Keeping the representation compact reduces the disk space utilization as well as I/Os time. Like WTOD and HST use two bits to identify a leaf and a rightmost child.

In the next chapter, we will present our compact binary suffix tree (CBST) index representation, which takes the advantages of current disk-based ST representations. We investigate efficient ways for constructing the CBST index. We then perform extensive experiments which illustrates that our CBST outperforms the other disk based ST technques in several ways including construction time, search, and scalability, for short as well as long sequences. The experiments will be presidented in chapter 4.

Chapter

3 CBST Representation and Construction Algorithm

In this chapter, we first introduce our proposed compact binary ST (CBST) index representation. Our proposed CBST takes advantage of existing disk-based ST representations. Then we introduce our CBST index construction algorithm, which is the improvement from B2ST. We introduce the CBST index construction algorithm for the chromosome-scale level sequences, and then extend it to genome-scale level and above. For ease of presentation, we introduce the CBST algorithm in two parts, called sorting and building, however, its actual implementation we developed is monolithic. An important feature of our implementation is that it is adaptive to the size of the input sequence, which takes into account the available main memory and decides the size of the partitions in order to reduce the construction time. We elaborate on this in the sequel.

3.1 Compact Binary Suffix Tree (CBST) Index Representation

Our CBST representation is a two-level index structure, which like to HST combines a lookup table (LT), called dividers in B2ST [Barsky et al., 2009]. Figure 9 shows this structure. The small, memory-resident LT serves as an index to the large, disk-resident

STs.

Once we add a suffix to the tree, we do not need to access it any more. Thus, we can save our CBST nodes to consecutive array elements in both memory and disk. This is the same as TDD, STTD64 and B2ST. Inside a binary suffix tree, all the branch nodes can have two children only, and leaves have no child. This allows using two child pointers for branch nodes. We can represent the entire suffix tree as a fixed size array for branch and leaf nodes. As such, we have the same number of tree nodes as before: the tree has one leaf and one branch node for each suffix inserted. Figures 10 and 11 show the data structure used in our CBST index to store the binary suffix trees (BSTs).

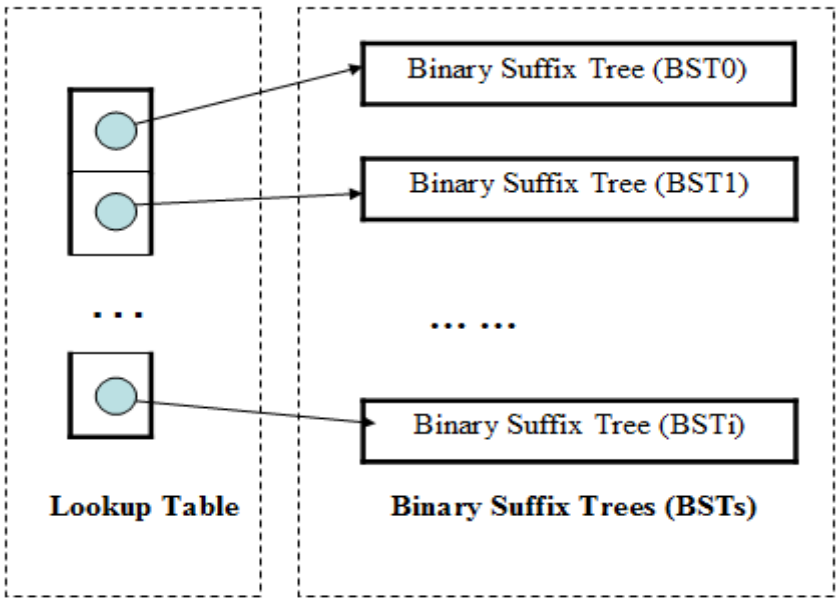


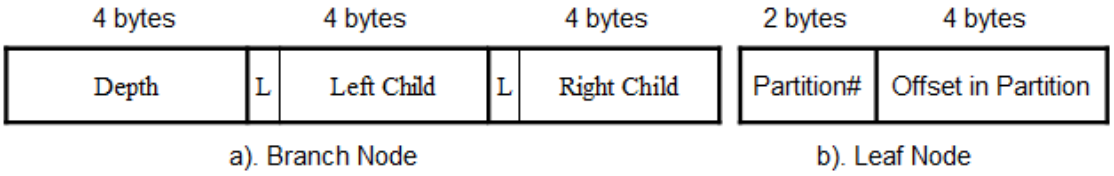
Figure 9. The two Levels of CBST Index Representation

```
typedef struct stBranch{     //Branch node, 4*3=12Bytes
    unsigned int deepth;     //longest lcp # can go to 4GB
    unsigned int child[2];   //highest bit=1 identifies a leaf node
                             //thus, maximum nodes #=2^31=2GB
}stBranch;

typedef struct stLeaf{       //Leaf node, 2+4=6B
    unsigned short int partition_id;   //supports 2^16=65536 partitions
    unsigned int start_position;       //supports maximum size=2^32=4G
                                       //thus, with partition_id, it goes to 2^48
}stLeaf;

/*
typedef struct SuffixTreeNode{   //18Bytes
    stBranch* stbranch;          //point to a branch node
    stLeaf* stleaf;              //point to a leaf node
}stNode;
*/
```

Figure 10. CBST Index Data Structure



('L' means leaf bit)

Figure 11. CBST Index Representation

In CBST index representation, a branch node has 3 parts, each with 4 bytes, in which we store the left, right child offset in the BST array, and depth information, respectively. Recall that the depth, as defined in the context of HST index, is the

length of the path (the number of symbols) from the root to its parent. The leftmost bit of the left or right child part identifies a leaf node when it is equal to 1; otherwise it identifies a branch node. Thus, each BST can store 2^{31} nodes, i.e., 2 billion nodes. In the following chapter we will show how to adjust this to desired size of the output BST files.

For a leaf node of CBST index, we use 2 bytes to represent the "partition id" (in which the suffix can be located) and use 4 bytes to store the offset of the suffix inside the partition. This would allow CBST to represent 2^{16} partitions with up to 2^{32} suffixes in each partition. Thus, CBST can support input sequences of size up to 2^{48}, i.e., 256 terabytes in length. We remark that the CBST size could be further extended beyond this size by adding more bits to the leaf node representation and a slight change in the ST construction algorithm.

Thus, we have 18 bytes for each suffix being added to the ST. There is one leaf node and one branch node per inserted suffix. This means each tree node occupies 9 bytes on average. If CBST only needs to support one partition of size up to 4 gigabytes in length, as handled by TDD, HST and TRELLIS, we can delete the two bytes representing the "partition id". This makes the CBST index work with 8 bytes per node (similar to the HST representation). While TRELLIS representation occupies 36 bytes, B2ST occupies 48 bytes per suffix.

When writing the output of each BST to disk files, we update the LT index by storing the file name of the BST on the disk and a predefined-length prefix of the largest suffix (the last added suffix) inside this BST. Thus, our LT index stores the references to each on-disk BST files. Once all the BSTs are stored to disk files, at the end, we write the LT index to disk. Figure 11 shows the layout of the CBST index layout on

disk for the BST considered in Chapter 2 for sequence S.

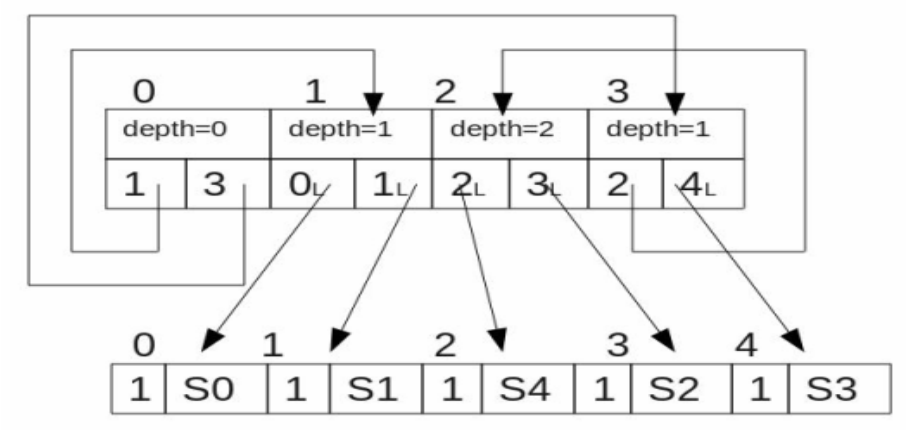


Figure 12. CBST Representation on Disk for Sequence S = ACGTG$

In Figure 12, the top array shows the branch nodes, while at the bottom it shows the leaf nodes array. Note that the partition id of every leaf node is 1 since there is only one partition. The number above each array element indicates its offset. The arrows indicate a branch node pointing to their corresponding children. The letter 'L' next to an offset means it points to a leaf node.

During query operations, the LT index is first loaded into the memory and stays there, while the BSTs will be loaded into the memory only when required. As illustrated in our experiments and results, this two-level structure exhibits good locality of reference for a disk-based ST index. Both HST and B2ST indexes considered this two-level index structure to speed up disk-based query operations

As explained above, our CBST index was inspired by HST and B2ST and took advantage of their features, but superseded both. The HST representation stores the depth information in a leaf node, which helped eliminate many back-jumps in search

operations, which in turn led to significant decrease in the number of disk I/Os required during search operations. Following that idea, we also store the depth information inside each branch node in CBST, while each leaf node stores the offset in a partition only. Both HST and CBST representations use a look-up table to locate the corresponding subtrees to reduce the number of disk I/Os required. The third feature borrowed from the others is the format of the data stored on the disk, i.e., the CBST representation also stores its branch and leaf nodes to the disk in consecutive array elements.

How to efficiently build our CBST index? Our CBST index construction algorithm can handle different size sequences, from several mega-bytes, like human chromosome Y (18MB) to several gigabytes, such as the whole 24 human chromosomes (3GB). That is, our proposed algorithm is suitable for chromosome-scale as well as genome-scale level sequences. In principle, CBST can handle huge input sequences of size up to 256 terabytes.

3.2 The CBST Algorithm for Chromosome-Scale Sequences

As mentioned earlier, we can convert each suffix tree into a suffix array (SA) in linear time by a depth-first traversal of the suffix tree. We can also convert the suffix array into a suffix tree in linear time, provided that the suffix array is augmented with the LCP length information between consecutive suffixes in the suffix array [Cameron, 2006]. We build our CBST index by first building the SAs.

For building a ST from sorted suffixes, we incrementally add the next suffix by

comparing it to the last added suffix alphabet by alphabet, in order to identify the storing location in the tree. This could be done easily for a binary suffix tree construction, since we only need to consider 0 and 1 branches. Due to the suffixes are sorted, the current suffix will be larger than the last one added, for which the first bit after the LCP length information (The prefix with the LCP length of the two suffixes are the same) should be '1' (1>0). Thus, we can follow the path from the root to the last suffix added in the tree, and traverse down for the LCP length information, until we get to the storing location. We then insert a new branch node and move the sub-tree below it to its left child, and add the current suffix to its right child. These steps are formally expressed in the construction algorithm shown in Figure 13.

The CBST algorithm for chromosome-scale includes two main steps: sorting and tree building. The longest human chromosome sequence is chromosome 2, with about 250MB. Thus, with a 2GB RAM in typical desktop computers today, we load each one of the human chromosomes into the main memory, sort them, and then obtain the suffix array. The remaining memory is used as an output buffer for building STs. For sorting the suffixes, we use an efficient sorting algorithm, called *MSufSort* [Michael and Simon, 2008], discussed in the following section 4.2. Once, all the suffixes are sorted, we add the sorted suffixes to the output buffer one by one and incrementally build the BSTs. Once the buffer is full, we flush it to the BST files on disk and update the LT index at the same time. We then re-initialize the buffer and start building a new BST again until all the sorted suffixes are dealt with. This creates on disk a collection of balanced tree files, all of which, except the last one, are of the same size as the output buffer. Finally, we write the LT index to a disk file. Following the presentation of the algorithm, we use a short sequence to demonstrate how the CBST algorithm works.

```
Algorithm buildCBST (Sequence: S, Suffix Tree: BST, Lookup Table: LT) {
//Phase I: Sort suffixes and collect: SA (S[0..(N-1)]), LCP(i,i+1) for 0<=i<N
1. Allocate memory buffer for sorting
2. Load S to the memory buffer from disk
3. Sort all suffixes of S under the buffer in lexicographical order
        //By using Msufsort algorithm, then we get the suffix array of S: S[0..(N-1)]
4. For (i=1; i<(N-1); i++) {
        //Collect longest common prefix length: |LCP(i-1,i)| of sorted suffixes S[i-1]
   and S[i]
5.      Calculate and record |LCP(i-1,i)|;
6.      Record the 1st bit of S[i];
7. } //end for

//Phase II: Build compact binary suffix tree (CBST)
8. Allocate output buffers: branchNodeB and leafNodeB;
    //Each branch node has left child[0], right child[1] and depth elements
9. Allocate memory buffer: boundaryPathB, to record boundary path;
    //Boundary path refers to all nodes from root to the last added suffix (leaf)
10. Add a new root node with depth=0 to branchNodeB and boundaryPathB;
11. For (i=0; i<(N-1); i++) {   //add the sorted suffix S[i] to BST
12.     If (only the root node in the output buffers) { //1st suffix to add to BST
13.         Add S[i] to leafNodeB;
14.         If (1st bit of S[i] == 0)
15.             Update child[0] of root to point to S[i];
16.         Else
17.             Update child[1] of root to point to S[i];
18.         Save S[i] to boundaryPathB;
19.     } //end 1st suffix to add to BST
20.     Else if (|LCP(i-1,i)| == 0) {    //S[i] be the right child of root
21.         Add S[i] to leafNodeB;
22.         Update child[1] of root to point to S[i];
23.         Save S[i] to boundaryPathB;
24.         If (one of output buffers is full) {
```

```
25.                 Flush output buffers to a disk BST file;
26.                 Save the prefix of S[i] and the BST filename to LT;
                    //Start a new BST in output buffer
27.                 Reset branchNodeB and boundaryPathB with only root node inside;
28.             }
29.         } //end S[i] be the right child of root


            //Assume two nodes in boundaryPathB: node[up] and node[down];
30.         Else if (depth of node[up] < |LCP(i-1,i)| <= depth of node[down]) { //on edge
31.             Add S[i] to leafNodeB;
32.             Add a new node between node[up] and node[down]: nodeNew with
                        depth=|LCP(i-1,i)| to branchNodeB;
33.             Update child[1] of node[up] to point to nodeNew;
34.             If (1st bit of S[i] == 0) {
35.                 Update child[0] of nodeNew to point to S[i];
36.                 Update child[1] of nodeNew to point to node[down];
37.             }
38.             Else {
39.                 Update child[1] of nodeNew to point to S[i];
40.                 Update child[0] of nodeNew to point to node[down];
41.             }
42.             If (one of output buffers is full) {
43.                 Same as lines 25 to 27
44.             }
45.         } // end on edge
46. } //end for
47. Flush output buffers to a disk BST file;
48. Save the prefix of S[i] and the BST filename to LT;
49. Save LT to disk file;
} //end buildCBST
```

Figure 13. The Pseudcode for CBST Construction Algotithm

In the above algorithm, the *boundary path* refers to the path from the root to the last added suffix on the BST.

Let us use the sequence S=ACGTG$ as an example to illustrate the CBST index construction algorithm. There are 5 suffixes in S, which we call as S0, …, S4. After sorting, we can easily get the SA of S as: [0, 1, 4, 2, 3]. Let us convert S to a binary sequence B=0001101110$ by using A=00, C=01, G=10, and T=11. Table 1 below shows every sorted suffix and its binary representation, the binary LCP length between any two adjacent sorted suffixes, as well as the first bit of a suffix after the LCP length.

Table 1. SA with LCP Length Information for Sequence S

SA#	Suffix	Binary Suffix	The LCP Length in binary	The first bit after LCP length
S0	ACGTG	0001101110	*	0
S1	CGTG	01101110	1	1
S4	G	10	0	1
S2	GTG	101110	2	1
S3	TG	1110	1	1

Our CBST index representation and the corresponding index construction algorithm are based on binary representation of the input sequence. However, in our implementation, we do not convert input sequence into binary. Instead, we take advantage of the efficient BST construction technique [Barsky et al., 2008]. In our illustrative example below, we use binary form for ease of presentation. In order to build BST, we only need to keep the LCP length information in binary format and the first bit after the LCP length. For example, for DNA sequences, we need 2 bits to represent the 4 alphabet letters (‘A’=00, ’C’=01, ’G’=10, ’T’=11). We thus can get the

LCP length in binary in two situations only:

1. The LCP length in binary is twice the original LCP length, if the next two letters after the original LCP length are either ‘G’ (=10) and ‘C’ (=01), or A (=00) and T (=11), for the two consecutive suffixes.

2. The LCP length in binary is twice the original LCP length plus 1, otherwise.

We start to build the BST in the output buffer from a root with no left and right children. The suffix S0 is the smallest one among all the suffixes of S, which we add to the output buffer. Since its first bit is ‘0’, we add S0 as the left child of the root. See Figure 14(a).

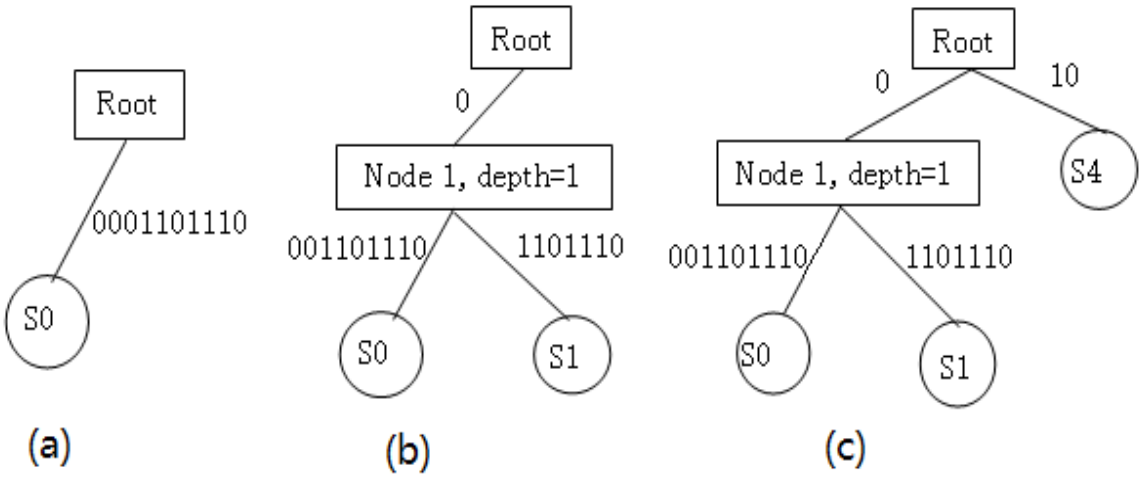


Figure 14. Construction of BST: (a). Adding S_0; (b). Adding S_1; (c). Adding S_4

Then, we add the next suffix S1 to the tree. As the LCP length in binary between S0 and S1 is 1, we traverse down from the root along the edge to the suffix S0, and count 1 bit to divide the edge between the root and S0, and add a new internal node, Node 1.

As the second bit of S1 is '1', we add S1 to be the child[1] of this new internal node, and move the sub-tree below this dividing point to its left child. The BST will be updated as Figure 14(b).

Next, we add S4. Since the LCP binary length between S4 and S1 is 0, counting down from the root, S4 is determined to be the child of the root. Since the first bit of S4 is 1, we add S4 as child[1] of the root, shown in Figure 14(c).

Same for adding S2 and S3. Figure 15 shows the tree after adding S2. After adding the last suffix S3, the final BST is the one shown in Figure 3 in Chapter 2.

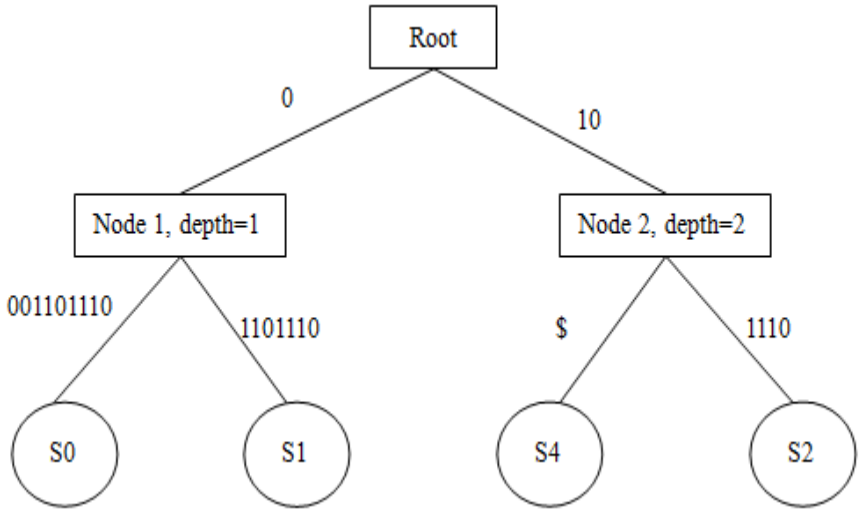


Figure 15. The BST Index After Adding the Suffix S_2

For the CBST construction algorithm, after all the suffixes are sorted in lexicographical order, we start by adding the smallest suffix in the buffer. For each step, we keep the path from the root to the last suffix added suffix to the tree. This path, called the *boundary path* (Refer to the figure 13), is the path in the suffix tree which corresponds to the largest lexicographical suffix being added [Barsky et al.,

2008]. Then following this path, we count down the LCP binary length along the edges and nodes, until we get to the inserting point. Proposition 1 below establishes the correctness of this process for building BST, the binary suffix tree. We remark that in order to identify the inserting point when building a ST for an alphabet different than binary, we need to calculate the LCP length for current suffix and all the remaining suffixes.

Proposition 1 [Barsky et al., 2008]

Let S be an input sequence, BST be a binary suffix tree being built, S[i] be the last suffix added to the ST, S[j] be the next suffix after S[i] to add to the BST, and LCP[i,j] is the LCP between S[j] and S[i]. Then, the split edge for S[j] lies on the *boundary path* of the BST, which is the path (including all the edges, internal and leaf nodes) from the root to the last added suffix S[i].

Proof: Since S[i] is the last suffix added to the BST, i.e., S[i] is the sub-string of S at positions i to N, the boundary path of BST corresponding to S[i] covers all the prefixes of S[i] including the sub-string S(i,i+LCP[i,j]). As S(j,j+LCP[i,j]) = S(i,i+LCP[i,j]), S[j] must be on the boundary path of the BST (which shares the same |LCP(i,j)| length of the path from root as S[i]), which corresponds to S[i] from the root.

Once we identified the inserting point for the current suffix S[j] to be added, we have to compare the symbols between S[j] and the suffix tree after the length of LCP[i,j]. Normally, we need to compare S[j] with all the branches in the suffix tree in order to

decide to which child of S[i] we should add S[j]. However for a BST, we have only two branches (0 or 1 child) to consider. Thus, S[j] has to be the right child (that is, child[1[) of the insert node since S[j] is larger than S[i]. The only exception is when the LCP length is equal to the length of the last suffix added. In this case, we may need to add the current suffix to the left child (that is, child[0]) of the last suffix added if the first bit after the LCP length of the current suffix is 0.

3.3 Sorting Suffixes in the CBST Algorithm

Suffix sorting is a key step in numerous applications, and is defined as the task of listing all the suffixes of a string or sequence in lexicographic order. Notable examples of such applications in our context include construction of the suffix array data structure and the Burrows-Wheeler transformation (BWT) [Burrows and Wheeler, 1994]. The BWT technique provides lossless compression, used as the main idea in the development of popular tools such as bzip2 [Seward, 2011]. Suffix sorting is also the main bottleneck in our CBST algorithm. Based on our experiments, this step takes more than 80% of the total index construction time. Thus, in order to speedup the construction, we need to have a fast sorting algorithm. At the same time, the suffix sorting algorithm should also be “lightweight”, that is, should require small memory space. The B2ST algorithm uses the Qsufsort [Larsson and Sdakane, 1999] for sorting the suffixes. In our CBST algorithm, instead, we decided to adopt and use the Msufsort technique [Michael and Simon, 2008] to sort indexes. This decision was based on the report in [SACA_Benchmarks, 2011]. We next compare these two suffix sorting techniques.

Qsufsort [Larsson and Sdakane, 1999] is one of the most efficient implementations of suffix sorting algorithm which uses the prefix-doubling technique. Qsufsort performs several rounds and at each round, it adopts the ternary-split quicksort (TSQS) technique proposed in [Bentley and Mcilroy, 1993]. The space complexity of Qsufsort is 8N, for an input sequence of size N, and its time complexity is O(NlogN) in the worse case.

In implementation of the CBST algorithm, we initially used Qsufsort for suffix sorting, however,we found three disadvantages. First, it requires the input sequence to be integers, that is, we needed an extra step to convert a DNA sequence into a number sequence. The second disadvantage of Qsufsort is its large space requirement (8N), which implies, compared to other techniques such as Msufsort [Michael and Simon, 2008], we can sort fewer suffixes using the same amount of main memory. We elaborate more on this in the next chapter, but at this point we should mention that the ability of sorting larger number of suffixes using the same amount of main memory will result in creating fewer number of partitions, and hence improved index construction. The third disadvantage of the Qsufsort algorithm is its speed, which we found to be slower than Msufsort [Michael and Simon, 2008]. The Msufsort [Michael and Simon, 2008] suffix sorting algorithm is shown to be a very space efficient and fast technique on [SACA_Benchmarks, 2011]. It manipulates the inverse suffix array (ISA) rather than the SA, where ISA is defined as the array ISA[j]=i, iff SA[i]=j. Thus, ISA provides the lexicographic rank of all the suffixes. It groups together all the suffixes having the same first character to form chains of suffixes, called *bucket*. Then it starts to assign ranks to the suffixes in each bucket in lexicographical order, and then use these ranks subsequently to speed up the assignment of ranks to the other suffixes. When the algorithm completes, every suffix has been assigned a unique rank,

based on its lexicographical order. Finally, it converts ISA to SA. The core of MSufSort is an efficient bucket sorting regime, called induction sorting [Michael and Simon, 2008]. The Msufsort algorithm is also called "lightweight" since it only needs (4+Z)×N working space, where Z is the number of bytes required per input symbol. For a DNA string, using one byte to represent each of the letters {A, C, G, T}, Msufsort requires only 5N bytes. The time complexity of Msufsort is $O(N^2 \log N)$ [Simon, 2005].

Table 2. Comparison between Qsufsort and Msufsort

Human Chromosomes	Sorting Time (Seconds)	
	QsufSort	MsufSort
Chr19 (56MB)	12	8
Chr9 (112MB)	32	23
Chr6 (171MB)	90	51
Chr1 (238MB)	111	62
Space requirement	8*N bytes	5*N bytes
Time complexity	O(n log n)	$O(n^2 \log n)$

In order to compare the two algorithms for sorting speed for the real DNA sequences, we test them on a Lenova ThinkStation 4220 with Intel(R) Xeon(R), CPU X3450 @ 2.67GHz, 2GB RAM, and 8192 KB cache size. Table 2 shows the results for the two sorting algorithms for real DNA sequences of different sizes. The results clearly indicates that on all these sequences, Msufsort outperforms over Qsufsort. In terms of space requirements, Msufsort is more space efficient for requiring only 5N bytes, as opposed to 8N required by Qsufsort. As will be explained, using the suffix sorting algorithm Msufsort in the CBST index construction technique, we can create fewer number of large partitiones, which in turn results in increased efficiency.

3.4 The CBST Algorithm for Genome-Scale Sequences and Larger

Our index construction technique is suitable for short sequences as well as very long sequences, such as genome-scale sequences and beyond, even when the input sequence is larger than the size of the available main memory. Recall that TDD, HST and TRELLIS algorithms require the input sequence to be in the main memory, and hence their capability to handle large sequences is limited by the available memory. In order to overcome this limitation to some extent, they resort to some compression method, using two bits to represent the DNA alphabet symbols. This allows them to build ST index for the entire human genome when the computer has 2GB RAM. This, however, is the largest sequence they can handle efficiently, or their efficiency reduces significantly for performing many random disk I/Os. The TRELLIS+ algorithm is an extension of TRELLIS which uses a buffering strategy for parts of the sequence, while the DIGEST algorithm adopts a method to buffer some fixed size prefix of each suffix. However, neither TRELLIS+ nor DIGEST can handle sequences larger than human genome-scale efficiently on a computer with 2GB RAM. Recently, the B2ST algorithm has been proposed, which partitions a long sequence, and then uses a sort-merge technique to sorts these partitions, and then merges them to BSTs. Our CBST algorithm for long sequences, introduced next, is an extension of the B2ST algorithm.

For the genome-scale and longer sequences, our CBST algorithm takes a “divide and conquer” approach and divides a long input sequence into short ones, called partitions, for which we can build SAs in the main memory. It then sorts the partitions and merges them to build the final BSTs on disk. Thus, our CBST algorithm performs

three main phases: (a). Partitioning the input sequence; (b). Sorting each partition; (c). Merging all the partitions to build the final trees, in the order mentioned. We next discuss details of each of these phases.

Creating partitions

We divide a long input sequence into a number of short sequences, called *partitions*. The size of each partition is determined by the amount of main memory available. In order to sort a partition and build its corresponding SA, we use the sorting algorithm Msufsort [Michael and Simon, 2008], which requires 5N bytes memory space, where N is the size of each partition. We also keep a small size of memory for an output buffer for building BSTs, then flushing to on-disk SA files once full. This is an additional step comparing to the chromosome scale algorithm introduced above in section 4.2 due to the large size of the input and the memory limitation, In our work using a typical desktop computer with 2GB RAM, we managed to sort a partition of about 330 MB.

When we divide the input sequence into partitions, we also add a short 'tail' to each partition, except the last one. And this 'tail' is not a substring of the partition. We get this 'tail' from the next partition, which is a short prefix of the next partition. This prefix serves as a sentinel to make sure all the suffixes in a partition are in the same order as they are in the original sequence. The following proposition from [Barsky et al., 2008] estabilishes this is necessary and sufficient. For two suffixes S[i] and S[j] of a big sequence S, we use the notation S[i] <= S[j] to mean S[i] is smaller or equal to S[j] in lexicographical order.

Proposition 2

Let S be a long sequence, P_k be the 'k'th partition of S (not the last partition), t be the 'tail' of P_k, which is a short prefix of the partition $P_{(k+1)}$ and is not a substring of P_k, S[i] and S[j] be the suffixes of S starting at positions i and j, respectively, and P_k[i] and P_k[j] be the suffixes of P starting at i and j (globally). P_k[i]·t means the concatenation P_k[i] and the tail 't'. Then, the concatenation P_k[i]·t<P_k[j]·t if and only if S[i]<S[j], here the sign '<" means lexicographically smaller while comparing two strings.

Proof: The *Only if* part. This is straightforward.

The *If* part. Without loss of generality, let us suppose i<j, i.e., the length of P_k[i] is larger than P_k[j]. As the tail 't' is not a substring of partition P_k, P_k[j]·t cannot be a prefix of P_k[i]·t. Let us assume the LCP length between Pi·t and Pj·t is L, and suppose C(i+L+1) and C(j+L+1) are the first letter of P_k[i] and P_k[j] after the LCP length, which must be different. Clearly, if C(i+L+1) > C(j+L+1) (or <) lexicographically, then Si > Sj (or <).

The partitioning phase is presented in Figure 16. The short tail guarantees the global order of the suffixes in each partition. We can determine it by keep reading from the next partition until we get a unique one for the current partition. If we cannot find it, we need to increase the size of a partition. In our experiments, we found that, for DNA sequences, the tail length does not exceed 125 characters when the partition size is about 150 MB. This explains why the tail is a "short" prefix.

```
Algorithm setPartitions (Sequences:S, Number of Partitions: K) {
1. Determine the partition size by M=|S|/K;
2. Set tail initial length = LEN;
3. Set tail incremental step = STEP;
4. For partition[i] (0<=i<K-1) {
5.      Load M symbols of S for partition[i]
6.      Load the consecutive M symbols of S for partition[i+1]
7.      tail = LEN prefix of partition[i+1]
8.      While (tail is a sub-string of partition[i]) {
9.              tail = tail + STEP
10.     }
11.     Concatenate tail to partition[i]
12.     Output partition[i] with its tail
13. } //end for
14. Output partition[K-1]; //The last partition
   } // End setPartitions
```

Figure 16. The Pseudocode for Partitioning Phase

Sorting Phase

In this phase, CBST performs two steps. Firstly, it sorts each partition and outputs its corresponding SA to a file on the disk. As the second step, CBST loads every possible partition pairs sequentially to the memory, and collects the longest common prefix (LCP) length with the order information for all the suffixes inside the partition pair. At the same time, we also collect a fixed length of prefix of each suffix in order to refer to it during the merging phase, introduced next. The tails will be excluded from the output SAs. Thus, if we have K partitions, the CBST algorithm collects the

LCP length and order information for K(K-1)/2 partition pairs. For example, these pairs include partition 0 with 1, 2, ..., (K-1), then partition 1 with 2,3, ...,(K-1), until partition (K-2) with (K-1).

In the sorting phase, presented in Figure 17, our CBST algorithm differs from B2ST in three ways. First, we use the Msufsort algorithm, which is faster and more space efficient than Qsufsort used in B2ST. The second difference is that CBST sorts each partition separately, while B2ST sorts suffixes based on partition pairs. For this, the B2ST algorithm concatenates partition pairs and includes their tails. It then collects the SAs for each partition and the LCP length and order information for each partition pairs. As a result, B2ST needs to sort more partition pairs, which is the key factor responsible for increased construction time. For example, sorting partition pairs using a typical computer with 2GB RAM, the maximum partition size that can be handled would be 200MB (=2000MB / 5bytes per symbol / 2 partitions). Considering the memory allocation for buffers to collect the LCP length and the prefix of suffix information, the size of a partition which B2ST processes will be even smaller. In our initial experiments with B2ST, the maximum partition size it could handle in a 2GB RAM was 150MB. This leads to increased number of partitions and hence longer sorting phase, resulting in long index construction time. The last difference is that CBST avoids converting the original sequence to number and binary format, which saves an extra time for the index construction comparing to B2ST.

```
Algorithm sortPartitions (Number of Parition: K, Size of Partition: P) {
// Step sort suffixes in each partition
1. For each partition[i] ( 0<=i<K) {
2.        Load partition[i] and sort it using Msufsort algorithm;
3.        Output partition[i] to SA[i] on disk file;
4. }

// Step II collect LCP length and order of suffixes
5. Allocate input buffers: partitionB and suffixarrayB; // For partitions and their SAs
6. Allocate output buffer: outputB; //for LCP length information
7. For every possible partition pair partition[i] and partition[i+1] (0<=i<K) {
8.      Load partition[i] and partition[i+1] to partitionB;
9.      Load SA[i] and SA[i+1] to suffixarrayB;
10.     While (not all suffixes under suffixarrayB being added to outputB) {
10.         Calculate LCP length of two consecutive suffixes in suffixarrayB;
              //lexicographical order
11.         Record the order of these two consecutive suffixes;
12.         Save LCP length and order information to ouputB;
13.         If (outputB is full) {
14.             Flush outputB to disk file;
15.         }
16.     } //end while
17. } //end for
  } // End sortPartitions
```

Figure 17. The Pseudocode for Sorting Phase

At the end of this phase, for a sequence S with K partitions, we have collected the following two sets of data files on the disk:

- Each SA for K partitions, and the fixed length of prefix of each suffix;

- K(K-1)/2 LCP lengths and suffix order information for all partition pairs.

With the above information, we can get all the suffixes in global order. Then we can build the BSTs for the input sequence S using the algorithm shown in Figure 13, phase II. We do not need to load the entire input sequence into the main memory again. Actually, we will never need to access the input sequence again.

Merging Phase

In this phase, our CBST algorithm uses the external two-phase multi-way merge-sort technique (2PMMS) [Garcia-Molina et al., 1999]. This is the same technique used in the B2ST algorithm. Normally, the K partitioned SAs will need K input buffers and K(K-1)/2 LCP length and order information input buffers in memory, and another output buffer for building BST. Figure 18 shows the the merging phase.

Once all the buffers are initialized, CBST starts to fill the input buffers with SAs and the LCP length and order information. A competition will be run against the top suffix inside all the SA buffers. The winner (smallest suffix) will be added to the output buffer. CBST keeps re-filling any one of the input buffers if it is exhausted, until all the suffixes inside all the SAs have being added to the output buffer (or the final ST).

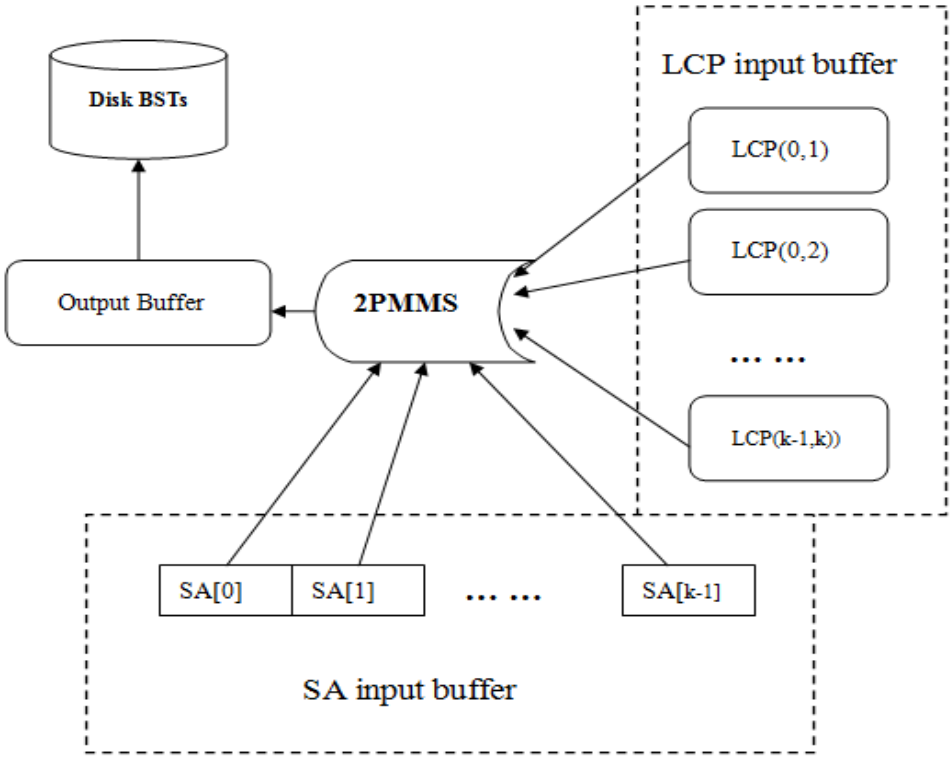


Figure 18. The merging in the CBST Index Construction Algorithm

Every suffix being added to the output buffer, CBST incrementally adds it to the BST using the algorithm presented as Phase II in Figure 13. Once the output buffer is full, it is flushed to a BST file on the disk. At the same time, CBST updates the LT index with the name of the files in which we saved BST and the predefined length prefix (refer to the sorting phase) of the last added suffix to this tree.

The whole merging phase will end after all the suffixes from all the SAs (all the partitions) are being added to the final trees. Finally, we output the LT index to a disk file also, which is an index to all the on disk balanced BSTs. Note, CBST algorithm accesses disk sequentially for both re-filling the input buffers and flushing output buffer to disk trees. Compared to random disk I/Os, the sequential scan saves

considerable time, as shown in our experiments. We present our experiments and results in Chapter 5.

3.5 Analysis of the CBST Algorithm

Our CBST algorithm is designed to support both short and long input sequences, and even sequences which do not fit in the main memory. It does not require the whole input to be resident in the memory. It also has very good locality of reference for tree construction. This is because the CBST algorithm performs sequential access to disk for both loading the input string and flushing the trees built. Given a sequence S with N symbols, we divide S into K partitions in order to build the SAs in the memory. In both partitioning and merging phases, the time complexity of CBST algorithm is O(N), that is, linear in the size of the input sequence. The time complexity of CBST for sorting is $O(N^2 \log N)$, for using the Msufsort suffix sorting algorithm. Thus, the overall running time of the CBST algorithm is $O(N^2 \log N)$, in the worst case.

3.6 Exact Match Algorithm Based on CBST Index

Exact match (EM) search is at the core of numerous exact and similarity (approximate) search applications in bioinformatics, and occupies 85% of the overall search time [Cameron, 2006]. In our work, we implemented a similar EM solution to the one proposed in [Halachev et al., 2005], which extends the memory based STEM technique to a disk based technique using buffering strategy. Figure 19 shows the EM algorithm that works based on the CBST index.

```
Algorithm EMsearch (Query Set: QS, Sequence: S, Index: CBST) {
// CBST index includes a lookup table (LT) and some binary ST files (BSTs)
// Phase I: Buffering the BSTs
1. Load QS and LT of CBST to memory from disk;
2. Sort QS in lexicographical order, get sortedQS;
3. For (each query Q in the sortedQS) {
4.      Use LT to determine a BST file including Q;
5.      If (the BST is not inside memory) {
6.          Allocate memory for buffering the BST;
7.          Load the BST from disk;
8.      }
9.      Else {
10.         Phase II: Find the answer_root for query Q;
11.     }
12. } //end for

//Phase II: Find the answer_root for query Q
13. Encode Q to binary B with the same rule used in CBST index;
14. q_length = number of bits in B;
15. answer_root = the first node of BST;
16. b_pointer = point to the first bit of B;
17. While (b_pointer doesnot reach the last bit of B) {//unexamined bits in B?
18.     If (b_pointer value = 1) { // if_1
19.         answer_root = its right child (child[1]);
20.         If(answer_root is a leaf) {
21.             Return (answer_root);
22.         }
23.     Else {
24          answer_root = its left child (child[0]);
25.         if(answer_root is a leaf) {
26.             return (answer_root);
27.         }
28.     } //end if_1
```

```
29.       if (depth of the answer_root = q_length) (
30.            return (answer_root);
31.        }
32.       Move b_pointer to the next bit of B;
33.  } //end while
34. Return (not found)

//Phase III: Verify answer_root from sequence S
35. L_offset = the 1st leaf of answer_root //by a top-down and depth-first traveling ST
36. Q_length = number of characters in the query Q
37. Get the sub-string SS of S with Q_length and at L_offset
38. If (P == SS) {
39.       Return (the offsets of all leaves of answer_root) //final answer
40. }
41. Else {
42.       Return (not found)
43. }
  } //end EMsearch
```

Figure 19. EM Search Algorithm On the CBST Index

The EM algorithm based on CBST index includes three components: buffer management, finding the answer node, and final verification. In first step, we only load the query set and the LT index in the main memory; the BSTs are read only when required. Once a BST is being loaded, we traverse the BST according to the query pattern bits (binary) until reaching a leaf or an internal node that its depth is equal to the length of the query pattern (in binary). We call the located leaf or the internal nodes are the answer node. Unlike traversing a traditional ST, the EM algorithm based on CBST index avoids comparing each character with the query pattern along the edges under a certain internal node. When the bit of the pattern is '1', then we continue searching on the right child subtree (or child[1]), otherwise on the left child

subtree (child[0]). In the last step, we verify the search result. If the answer node is a leaf node, then we simply verify it with the input. If the answer node is a internal node, we verify the smallest suffix (or the first leaf if traversing from top to down, left to right) under this answer node with the input. If it is the desired answer, then all the leaves of the answer node are query results.

We use a simple example to show how to find the answer root query pattern P='G' for the BST example in Figure 3. We list below the BST with traversing path in Figure 20. The arrows show the traverse path for the query pattern P.

By applying the same encoding rule as the CBST tree, the query pattern P in binary is equal to "10". We traverse the BST starting from the root to its right child of the root, then go to the left child of the branch node 3. We reach the answer root - the branch node 2. The leaves of node 2 indicates the final two suffixes are S4 and S2.

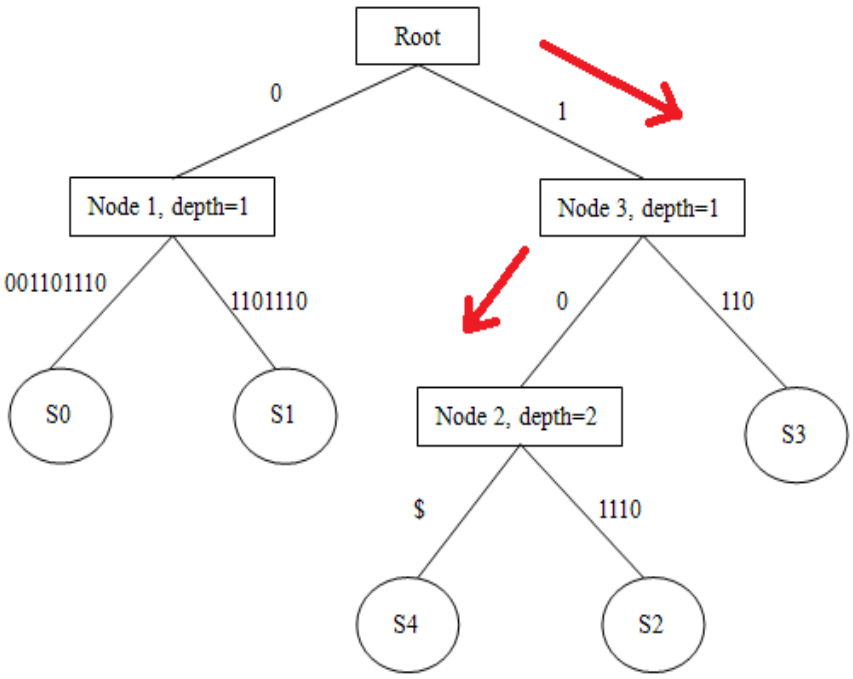


Figure 20. EM Searching for Query P='G'

3.7 Summary

In this chapter, we introduced the CBST index construction algorithm for sequences of a wide ranges of sizes, short to very long. We also developed an exact match search algorithm (EM) based on the CBST index. Our CBST algorithm is an extension of the B2ST algorithm, but as will be shown in the next chapter, CBST out-pereforms B2ST and requires less disk and main memory space, due to the compact representation of CBST index and the "lightweight" suffix sorting algorithm it uses. Compared to TDD, HST and TRELLIS, all of which require the entire input sequence to be in the main memory during the index construction, our CBST algorithm overcomes this restriction and avoids data skew problem. Moreover, CBST produces balanced and equal size BSTs in disk files.

The two level index data structure of CBST provides good locality of reference for disk based query processing. The disk-based trees are only loaded when required. Besides, the size of the BSTs is also the key issue for disk-based ST index. Given a query, a large size ST would mean long time would be required to load the tree into the main memory. Thus, another important feature of our CBST algorithm is that we can adjust the final size of the BSTs to the disk without increasing the index construction time and space. This is not true for other disk-based ST algorithms but the DIGEST and B2ST algorithms. Our extensive experiments and results obtained confirms all of the above. They are discussed in the following chapter.

Chapter

4 Experiments and Results

In this chapter, we evaluate the CBST index representation and its construction and search techniques. We compare CBST to the best known techniques in terms of the index construction time, the index size, and exact match (EM) search performance.

The B2ST [Barsky et al., 2009] is the most recent disk based ST construction algorithm. It has being reported to be the fastest among the disk based ST algorithms for sequences larger than the human genome. We consider it in our study when comparing the performance of techniques that can handle large sequences. The source code for B2ST was made available to us by the authors, for which we are thankful [Barsky, 2011].

Both TRELLIS [Phoophakdee and Zaki, 2007] and HST [Halachev et al., 2007] can be used for creating ST indexes for large sequences like human genome. Both techniques are shown to perform well for disk based search operations. We compare them with our CBST for individual chromosomes as well as genome sequences. We obtained the codes for both algorithms from the authors (the binary code of TRELLIS, and the source code of HST). However, we faced problem running the HST code for the second level index and hence consider STTD64 [Halachev et al., 2007], the first level of HST index, in our performance evaluation and comparison of index construction time and storage requirements.

We also include suffix array (SA) based techniques in our comparison. For this, we

consider Vmatch [Vmatch, 2011] which is a commercial software tool that implements enhanced SA (ESA) [Abouelhoda et al., 2004]. It is a memory based algorithm and can only support chromosome-scale level sequences under a typical desktop of 2GB RAM). There is a disk based SA index construction algorithm, called DC3 [Dementiev et al., 2005], which uses the pipe-line technique and needs multiple disks. It only constructs the basic SA and has been shown to be inferior to TDD [Tian et al., 2005] on a typical computer. Thus we only include Vmatch in our comparison for chromosome-scale level sequences, although it is not disk based. After authorized by the authors, we obtained Vmatch code from their web site [Vmatch, 2011].

We implemented our CBST algorithm in C++ and compiled under Eclipse Galileo Version 3.5.1 (build id: M20090917-0800) with optimization parameters "-O3 -Wall -c -fmessage-length=0. The code for Msufsort [Michael and Simon, 2008] was obtained from [Michael, 201]. We conducted all our experiments on a Lenovo ThinkStation 4220 with Intel(R) Xeon(R), CPU X3450 @ 2.67GHz, 2GB RAM, and 8192 KB cache size. The desktop runs Fedora release 13 (Goddard).

Next, we describe the DNA sequence data we used in our experiments. Then, we describe how to adjust the parameters for our CBST algorithm to get the best performance. After that, we compare the performance of CBST and other techniques, based on the construction time, storage space requirements, and the exact match (EM) performance.

4.1 Experiment Sequence Data

The sequences we used in our experiments include all the 24 human chromosomes,

downloaded from [NCBI, 2011], and the chimpanzee and zebra fish genomes from [USCS, 2011]. The chimpanzee DNA sequence has around 3.2 GB, while zebra fish has around 1.3 GB. Figure 21 shows the size of the 24 human chromosomes in MB.

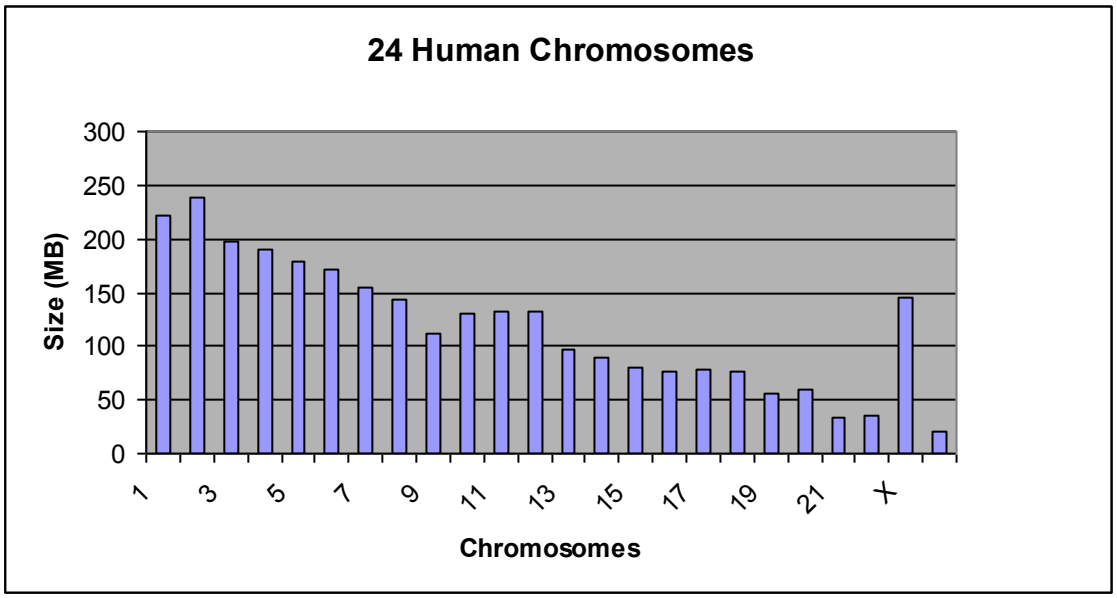


Figure 21. The 24 Human Chromosomes and Their Sizes

Our CBST algorithm is effective in handling short to very long sequences. In our performance study, we classify our experiments based on the size of the input sequences into 4 types, described as follows.

- Type 1: short sequences of size up to 250 MB. This includes each one of the 24 human chromosomes sequences;

- Type 2: medium size sequences up to 1 GB. For this, we consider the concatenation of the human chromosomes 1 and 2;

● Type 3: long sequences between 1 GB and 4 GB. For this, we concatenate all the 24 human chromosomes;

● Type 4: very long sequences with more than 4 GB. For this, we concatenate the 24 human chromosomes, chimpanzee, and zebra fish sequences.

4.2 Adjusting the Parameters for CBST Construction Algorithm

Our CBST index representation can support up to 256 terabytes long sequences in theory. It can adapt to the size of a given input sequence and the available main memory, and decide the number of partitions needed for the sequence. In our experiments, we found two key parameters that heavily influence the performance of the CBST algorithm: the number of partitions and the output buffer size. As the CBST algorithm needs to collect the LCP length information for any partition pairs, more number of partitions would mean more number of LCP length pairs information required to be sorted and collected. It would also mean more disk I/Os. While another key parameter of CBST algorithm is the size of the output buffer. As mentioned in Chapter 5 and 6, the CBST algorithm outputs balanced, equal size BSTs on disk (except for the last BST), which is defined by the size of the output buffer. We noted that the performance of search operations heavily relies on the size of the BSTs on disk. We show the best values for these two parameters of CBST algorithm based on experiments on our computer system setup.

Choosing the Number of Partitions

We first perform experiments on the "Type 2" data sequence for CBST algorithm with different sizes of partition but with a fixed size (50 MB) output buffer. Table 3 shows the results. As the partition size decreases and the number of partitions increases, the whole index construction takes more time. The construction time for the same sequence with 6 partitions is doubled compared to when considering only 2 partitions. Thus, under the main memory limitation, CBST needs to divide a sequence into few partitions as possible in order to gain fast construction. This indicates when more memory is available, the construction will require less time.

Table 3. CBST Construction on "Type 2" Data with Output Buffer of Size 50MB

#	Partition Size (MB)	Number of Partitions	Construction Time (Seconds)
1	330	2	480
2	190	3	550
3	120	4	679
4	100	5	850
5	80	6	974

In our experiments, we chose 330MB as the partition size based on our computer with 2GB RAM.

Choosing the Output Buffer Size

In order to find how the size of the BSTs influences the EM search performance, we run the CBST algorithm 4 times with different output buffer size for the "Type 3" data.

Then, we obtained the following 4 groups of different sizes of BSTs on the disk: 7MB, 10MB, 100MB, and 1GB. In each group, all the BSTs are of the same size (except for the last tree) since CBST produces balanced equal size of trees. For example, for the first group, all the trees produced are of size 7MB. We then performed the exact match (EM) search queries based on each group trees. We also created and used two query sets: Set-100 and Set-1000. Set-100 included 100 query patterns, while Set-1000 inlcuded 1000 query patterns. In each query set, we randomly extracted the query patterns from the human chromosome 2 with different lengths: 7, 11, 15, 41 and 91. For example, Set-100 with pattern length 7 included 100 queries and each query included a pattern with 7 symbols. Table 4 shows the results of these experiments.

Table 4. Query Results with Different Sub-ST Sizes for "Type 3" Data

Query Sets		**Query Time (Seconds) with Different BST Size (MB)**			
Query Set	**Pattern Length**	**1000MB**	**100MB**	**10MB**	**7MB**
Set-100	7	415	109	13	15
	11	383	72	9	10
	15	375	72	8	12
	41	385	73	8	11
	91	367	70	8	10
Set-1000	7	877	752	80	175
	11	475	369	73	92
	15	464	355	71	96
	41	462	310	72	99
	91	462	353	71	95

The results of our experiments indicate that when the size of the on-disk BSTs reduces

from 1GB to 100MB, we get a speed-up of 5 for processing the set of 100 queries, and speed-up of 1.5 for the set of 1000 queries. If we keep reducing the size of the BSTs to 10M, the EM search performance tends to improve, until the size of the BSTs reaches to 7MB. As the EM algorithm introduced in Chapter 4, for a query set, we load and keep the LT index inside the memory, while loading the BSTs only when required. Thus, the EM search performance is based on the disk access speed and the size of the BSTs.

In our experiments, we chose 10MB as the output buffer size for building the CBST index BSTs, and the EM search queries are based on this size of the BSTs.

4.3 Index Construction Time

Results for "Type 1" Sequence

We compare the performance of the construction algorithms of CBST with existing indexing techniques for chromosome-scale level sequences. These techniques include Vmatch, TRELLIS and STTD64. According to the "readme" file for TRELLIS source, we set its initial prefix length to 3 for the "Type 1" sequences. Figures 22 and 23 show the results for the 24 human chromosomes. The B2ST code we have worked only for the "Types 3 and 4" sequences. and hence we do not consider B2ST here for the "Type 1" category.

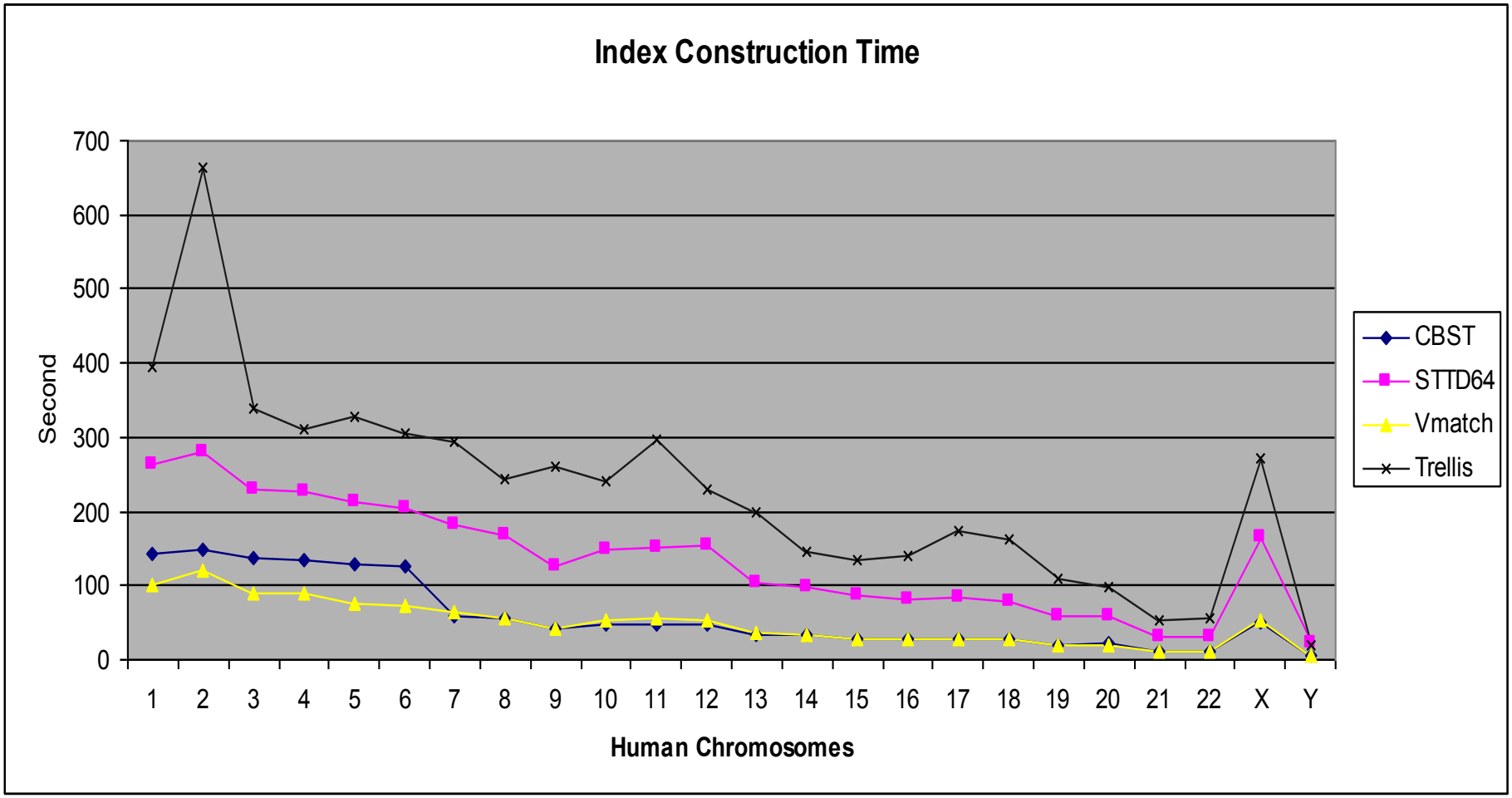


Figure 22. Index Construction Time Comparison

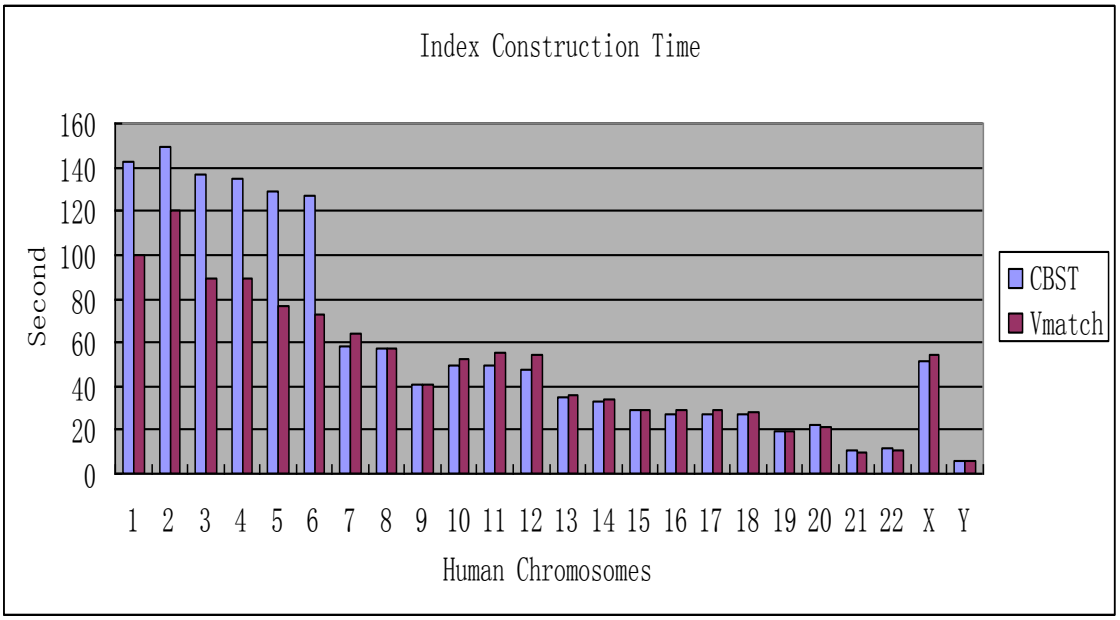


Figure 23. Comparison of CBST and Vmatch Index Construction Times

For the “Type 1” data, CBST is twice faster than STTD64 and three times faster than TRELLIS on each of the 24 choromosomes. This is mainly due to better locality of reference of CBST and the fast sorting algorithm it uses. Providing enough memory (2GB in our context), the STTD64 algorithm performs better than TRELLIS on short sequences. Recall that STTD64 loads a sequence into memory partitions it, and then loads each partition again to build the final suffix tree. That is, STTD64 performs 4 disk I/Os, while CBST performs only 2 disk I/Os. This is because for short sequences, CBST can load and sort a sequence to memory once, and then build BSTs incrementally in the output buffer; once the buffer is full, it is flushed to disk.

For index construction, the performance of CBST is close to Vmatch, which is memory based. When the input sequence is no more than 150 MB, Vmatch and

CBST show similar performance for index construction. However, the advantage of CBST is that it is adaptabile, in that knowing the size of the available main memory, it can decide the number of required partitions and hence adapts itself suitably to short and long sequences accordingly. the number of partitions required. For short sequences of the “Type 1”, CBST needs only one partition for a 2GB main memory. We next compare the CBST algorithm to others on long sequences.

Results for Sequences of “Types 2, 3 and 4” Sequences

Table 5 shows the performance of different index construction algorithms for sequences of the “Types 2, 3 and 4”. Due to STTD64 and TRELLIS can not support “Type 4” data sets, we use an ‘n/a’ in the table to indicate that the experiment could not be carried out. Memory-based Vmatch is not capable to these three types of sequences.

Table 5. Comparison of Index Construction Times for HST, B2ST and CBST

Data set	**Sequences**	**STTD64**	**TRELLIS**	**B2ST**	**CBST**
Type-2	Human Chr1 & 2 (total size:461MB)	10m19s	25m51s	21m18s	8m01s
Type-3	24 Human chromosomes (total size:2.85GB)	11h32m	4h30m	4h5m	2h57m
Type-4	24 Human chromosomes & chimpanzee & zebra fish genomes (total size:7.67GB)	n/a	n/a	13h12m	9h47m

CBST outperformes the disk based algorithms: TRELLIS, STTD64 and B2ST. Compared to B2ST, CBST has two advantages: (1) uses a fast and space efficient sorting algorithm and (2) produces compact BST representation (CBST). These two factors together result in reduced time for sequential I/Os. STTD64 is slow on “Type 3” data set, because it needs to keep the whole input sequence in the main memory for better locality of references. This in turn leaves less space for STTD64 to keep its dynamic buffers. We remark that in our experiments, B2ST index construction algorithm was slower than reported in [Barsky et al., 2009], perhaps due to different computer setups.

4.4 Index Storage Requirements

Normally, the size of a suffix tree index is linear in the size of the input sequence. Thus, In this section, we only compare the space requirements for ST index for “Type 1” and “Type 3” data, which are the chromosome-scale and genome-scale sequeces. We choose these two types of sequences due to Vmatch is only capable to chromosome-scale sequences and both STTD64 and TRELLIS are only capable to genome-scale sequences. The other types of data will get the similar results to these two data sets.

Index Storage Requirements for “Type 1” Sequences

Table 6 shows the storage requirement comparison for “Type 1” data. It includes both

the total index size and the average size of each 24 human chromosome. For the TRELLIS algorithm, we collect the data without the suffix links. TRELLIS takes more disk spaces If including the suffix links. The B2ST was coded only for the "Types 3 and 4" sequences and hence not included here.

Table 6. Index Storage for "Type 1" Data

Total Size (GB)					**Average Size (Byte per alphabet)**			
Chr#	**CBST**	**STTD64**	**Vmatch**	**TRELLIS**	**CBST**	**STTD64**	**Vmatch**	**TRELLIS**
1	4.26	2.86	2.77	5.74	19.12	12.83	12.43	25.78
2	4.56	3.05	2.96	6.12	19.14	12.82	12.45	25.71
3	3.80	2.54	2.47	5.11	19.17	12.85	12.47	25.80
4	3.63	2.43	2.36	4.56	19.17	12.85	12.47	24.05
5	3.41	2.29	2.22	4.59	19.16	12.84	12.46	25.78
6	3.27	2.20	2.13	4.41	19.16	12.85	12.46	25.81
7	2.96	1.99	1.93	4.00	19.09	12.82	12.42	25.77
8	2.75	1.84	1.79	3.69	19.16	12.83	12.46	25.74
9	2.14	1.43	1.39	2.87	19.06	12.77	12.40	25.62
10	2.50	1.68	1.63	3.37	19.14	12.82	12.44	25.73
11	2.54	1.71	1.65	3.43	19.16	12.85	12.46	25.81
12	2.54	1.70	1.65	3.42	19.14	12.85	12.45	25.83
13	1.85	1.24	1.21	2.48	19.17	12.81	12.47	25.67
14	1.70	1.14	1.10	2.29	19.15	12.84	12.45	25.77
15	1.52	1.02	0.99	2.04	19.07	12.78	12.40	25.65
16	1.45	0.98	0.95	1.98	18.89	12.78	12.38	25.70
17	1.47	0.99	0.96	2.00	18.99	12.77	12.35	25.71
18	1.45	0.97	0.94	1.94	19.17	12.80	12.46	25.63
19	1.07	0.72	0.69	1.47	18.95	12.88	12.32	26.13
20	1.16	0.77	0.75	1.55	19.13	12.80	12.44	25.67
21	0.65	0.43	0.42	0.87	19.10	12.76	12.42	25.54
22	0.66	0.44	0.43	0.89	19.04	12.80	12.39	25.75
X	2.64	1.78	1.72	3.59	18.11	12.21	11.78	24.62
Y	0.36	0.24	0.23	0.49	18.13	12.27	11.80	24.78

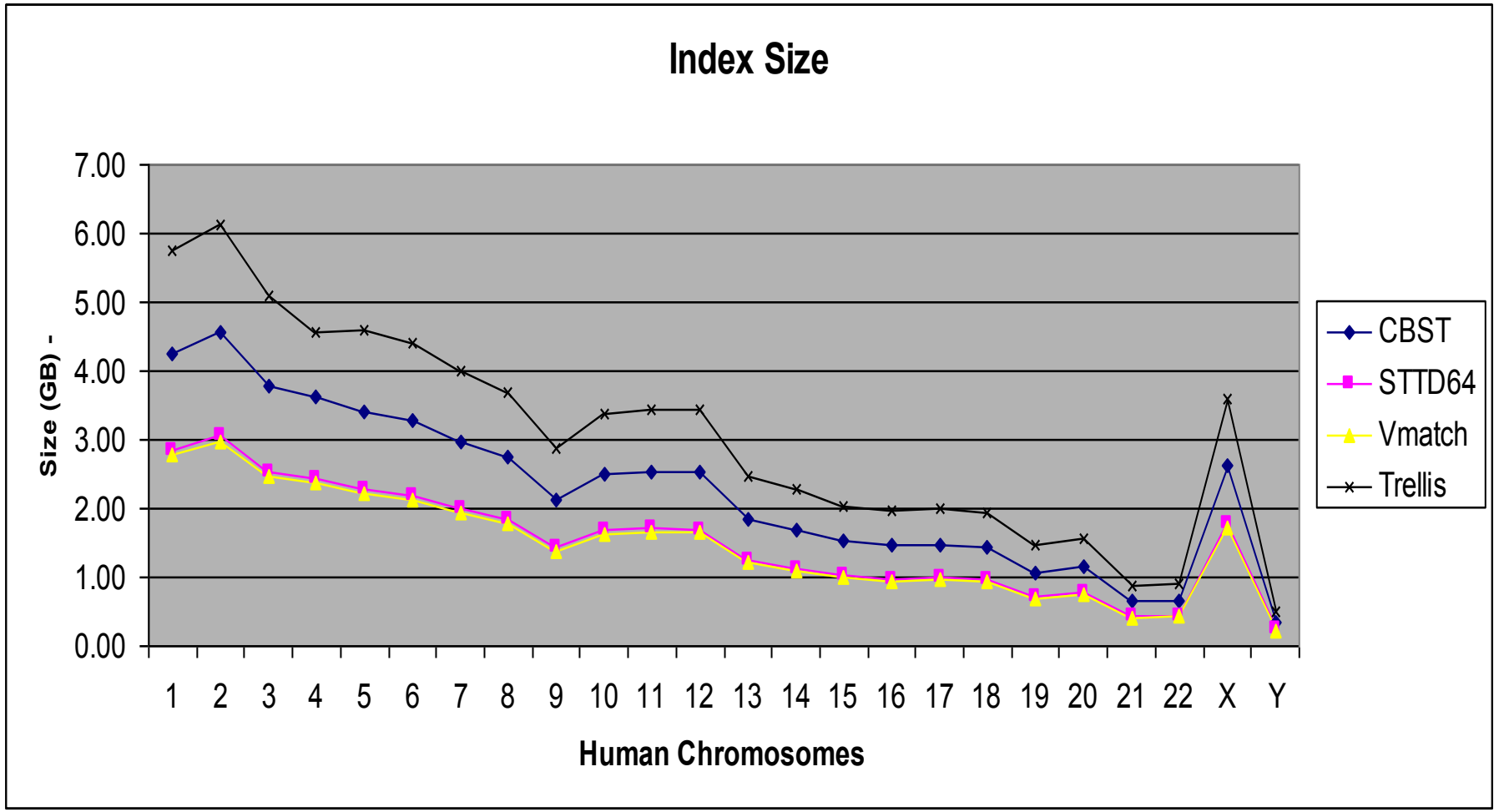


Figure 24. Index Size for “Type 1” Sequence Data

The graphs in Figure 24 conveniently present the index size data shown in the table above. As can be seen, STTD64 takes the least amount of storage space among disk based ST, which by the way is almost the same as Vmatch. On average, STTD64 needs around 12.8 bytes per alphabet symbol. The TRELLIS index occupies more space than the others -- about 25.5 bytes per alphabet symbol on average. CBST lies between TRELLIS and STTD64, requiring about 19.13 bytes per symbol. However, for “Type 1” sequences, only one paritition required in CBST index, we can save one byte for each symbol by deleting the byte that represents the partition id in the index representation (Refer to index construction section in Chaper 4). This means, for “Type 1” data, CBST needs on average 18.13 bytes per alphabet symbol.

Index Storage Requirements for “Type 3” Sequences

To determine the storage requirements for “Type 3” sequence data, we collect the size of the final index and the size of all intermediate temporary data required in building the STs for all the algorithms. Same as for ‘Type 1’ data, we do not consider the size of the suffix links used in TRELLIS. Table 7 shows the results.

Table 7. Index Storage Costs for “Type 3” Sequence Data

Size \ Algorithms	TRELLIS	STTD64	B2ST	CBST
Final ST index size (GB)	71.86	34	122	52
Intermediate data size (GB)	0	21	236	107
Average (byte per character)	25.22	11.93	42.82	18.25

On average, STTD64 requires the least storage space, for an average of 11.93 bytes per alphabet symbol. However, it can only support up to 4GB long sequences. For such size sequences, TRELLIS requires 25.22 bytes per symbol, CBST requires 18.25 per symbol, and B2ST requires more space than all others. B2ST requires 42.82 per symbol. Compared to CBST, the space required by B2ST to keep intermediate temporary data is doubled as it needs to encode the input sequence to numbers for sorting and to binary format for merging.

All of the above indexing algorithms for sequences produce a forest of STs. The largest ST produced by STTD64 is 1.447GB, while the smallest one is 83MB. As mentioned before, in this case, a query that uses this largest tree results in many random disk I/Os on a computer system with less than 1.4GB RAM. However, both CBST and B2ST produce equal size BSTs. And the size is equal to the size of the output buffer, which is adjustable in CBST, depending on the the available main memory and the query requirement.

4.5 Exact Match (EM) Search Performance

We also evaluated the exact match (EM) search performance of the CBST algorithm and compared it with existing techniques. For this, we used two datasets. For short sequences like “Type 1” data, we compare our algorithm with Vmatch, a memory based indexing technique which implements enhanced suffix array. For long sequences like “Type 3” data (TRELLIS is not capable to “Type 4” sequence), we compare CBST with TRELLIS and B2ST. To ensure that caching is not playing a role and the results of previous queries are not reused, we had a “cold strat” for our

experiments, i.e, the memory was purged before each run.

For easy of presentation, we use q_x_y to denote a query set that includes x number of queries with a query pattern length y. For example, q_7_100 refers to a query set which includes 100 queries, each of length 7 (7 symbols). As done in [Halachev, 2009] and [Abouelhoda et al., 2004], each query pattern in a query set we used in our experiments is randomly extracted from human chromosome 2 and its reverse.

EM Search Operations for "Type 1" Sequence Data

Figures 25 and 26 show the results of EM search performance for CBST and Vmatch algorithms with query sets of 100 and 1000 queries, respectively. Each value shown in these figures is the average for all the 24 human chromosomes in "Type 1" sequences data.

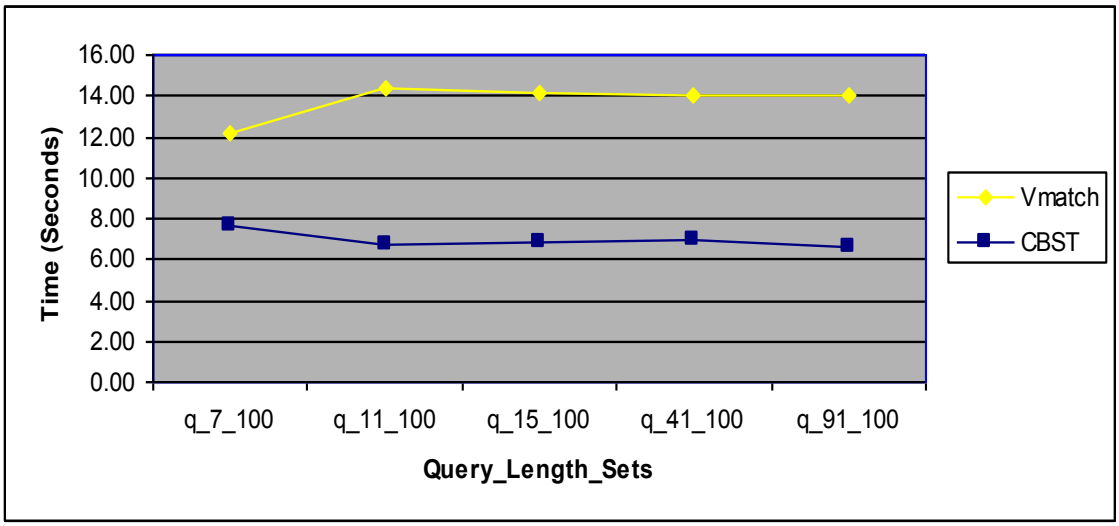


Figure 25. EM Search Performance on "Type 1" Data with 100 Queries

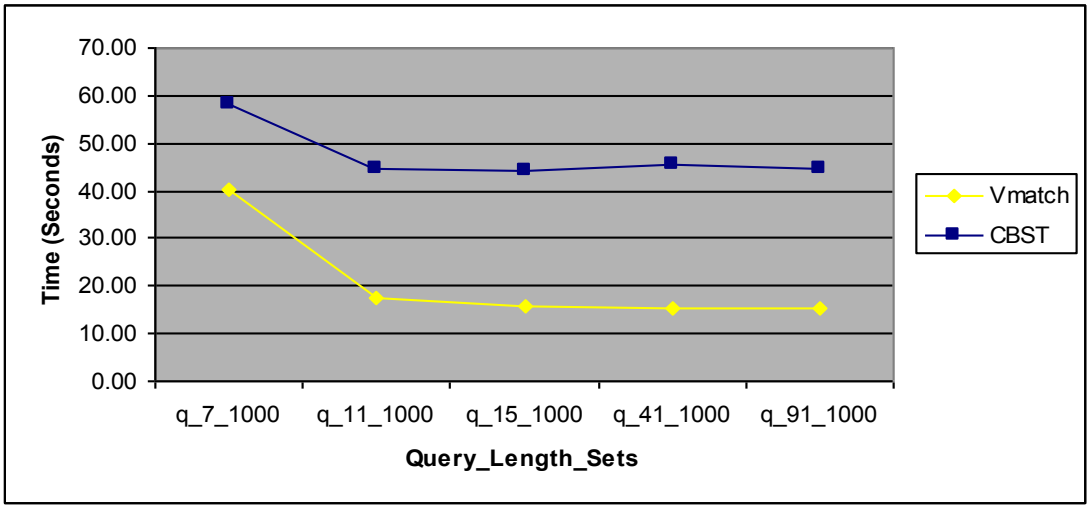


Figure 26. EM Search Performance on "Type 1" Data with 1000 Queries

From the above two figures, it can be seen that CBST is twice faster than Vmatch for the set with 100 queries, while it is slower for 1000 queries set. An explanation for this is follows. Since Vmatch is a memory based algorithm, it needs the whole index to be present in the memory for it to perform, and this is done only once during the processing of all the queries in the set. On the other hand, CBST is a disk based index which loads the sub-tree indexes into the main memory only when required. Thus, when the query set is larger, Vmatch is at the advantage and hence faster. However, Vmatch is not suitable for long sequences, including "Type 2" data. In what follows, we thus compare CBST with disk based index which can handle long sequences.

EM Search Operations for "Type 3" Sequence Data

As mentioned above, both CBST and B2ST are capable for handling long sequences, while TRELLIS can only support sequences of size up to 4GB. In order to compare the search performance of these disk-based indexes, we consider "Type 3" sequences in this set of

experiments. Figures 27 and 28 show the results. Note, the results here for the TRELLIS algorithm are based on its ST index with suffix links.

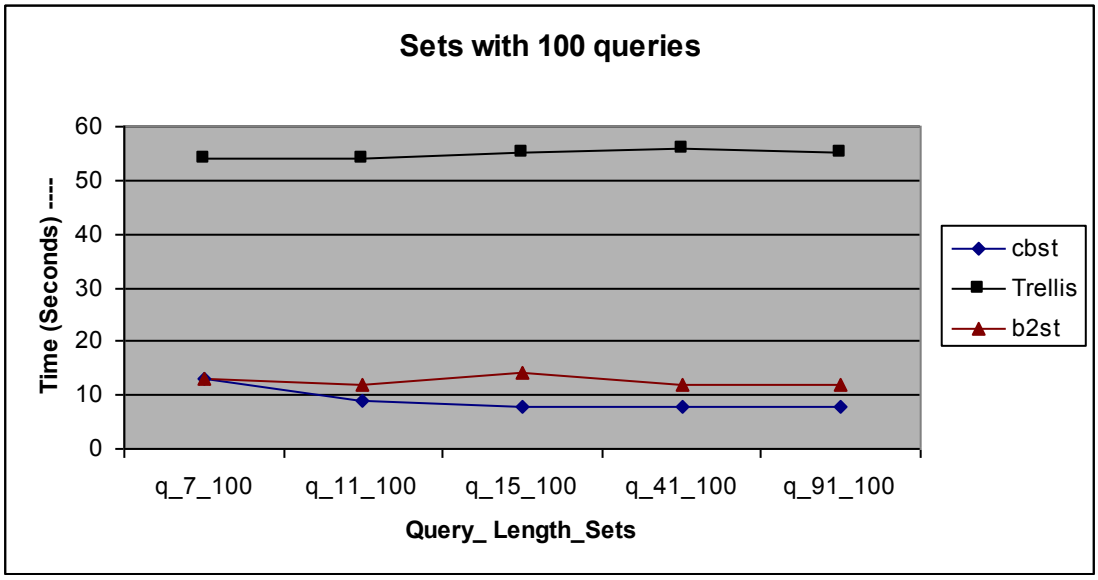


Figure 27. EM Search Time (in seconds) on "Type 3" Data with 100 Queries

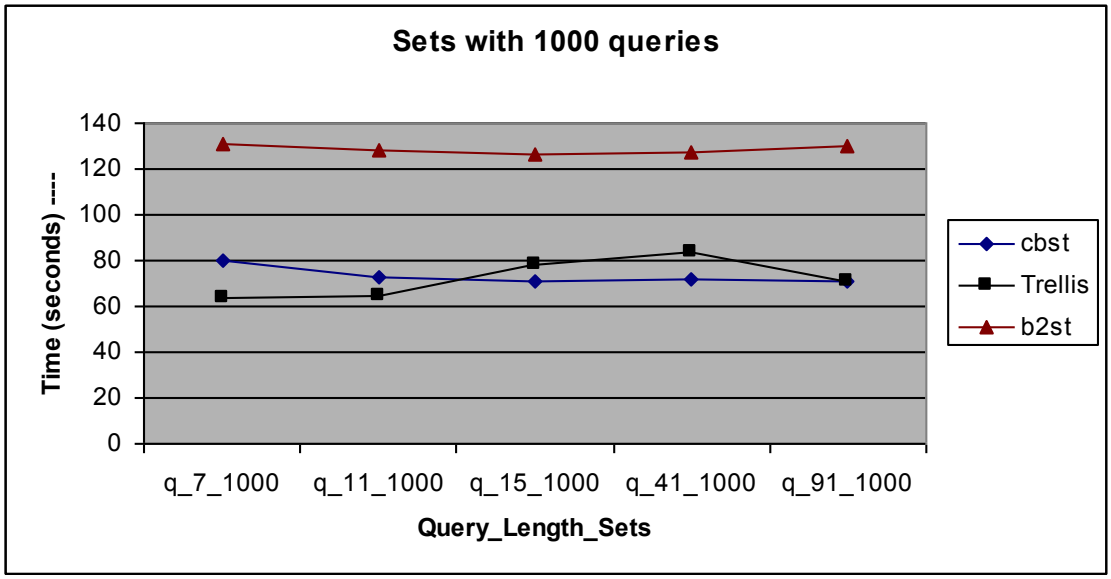


Figure 28. EM Search Time (in seconds) on "Type 3" Data with 1000 Queries

For long sequences, our results indicate that the EM search performance by CBST shows better performance, compared to TRELLIS and B2ST. For the smaller dataset of 100 queries, both CBST and B2ST are 5 times faster than TRELLIS. When the number of queries is 1000, the efficiency of TRELLIS catches up with CBST due to TRELLIS' suffix link advantage. In this case, both CBST and TRELLIS are 2 times faster than the B2ST algorithm. For large number of queries, we believe the EM search operations based on CBST is more advantageous over B2ST for its buffering strategy.

4.6 Summary

In this chapter, we evaluated the performance of the proposed CBST index. Our results indicated that CBST is a desired choice as it is suitable for indexing a wide range of sequences, from short to very large sequences. This capability is due to its design being parametric, making it suitable to handle any sequence size, effectively and efficiently. It can be easily configured and adapted based on the available main memory size to decide the possible largest partition size, which in turn results in increased efficiency in index construction. We also explained how to decide the two key parameters of the CBST algorithm in order to gain the best performance in index construction and exact match search tasks.

For index construction, our CBST is 2 times faster than STTD64 and 4 times faster than TRELLIS, for short sequences. It almost enjoys the same efficiency as the memory based algorithm Vmatch. For large sequences such as the entire human genome, CBST is 1.3 times faster than B2ST, 1.5 times faster than TRELLIS, and 2.8

times faster than STTD64. CBST is moderate in spare requirement by being between STTD64 and TRELLIS, and requires much less space than B2ST. However, both STTD64 and TRELLIS can support sequences of size up to 4GB, while CBST can support up to 256 TB long sequences.

We also compare the EM query operations based on the final indexes they generate. For short sequences and with small number of queries, CBST outperformes Vmatch. For disk based STs, CBST performs at least twice faster than TRELLIS and B2ST.

The CBST algorithm outputs a forest of balanced, equal size ST files on disk. Furthermore, our index can adjust the output buffer size in order to produce tree files of different sizes according to the query requirements. Based on our experiments and the results presented in this chapter, we conclude that our CBST algorithm is a desired efficient and scalable disk based ST technique. We also conclude that CBST is the fastest disk-based ST index construction algorithm so far.

Chapter

5 Conclusion and Future Work

The amount of biological sequence data is growing exponentially. To analyze such large amount of data, time and space efficient methods are necessary. In this thesis, we studied existing external suffix tree (ST) indexing techniques, and proposed a new disk-based ST index representation on binary alphabet, called *compact binary ST (CBST)*. We also introduced an efficient index construction and exact match (EM) search algorithms based on the CBST index. The results of our extensive experiments and their analyses clearly indicated that the proposed indexing technique outperforms existing ST techniques.

WOTD is one of the most space-saving memory-based ST index representations. TDD and HST extend WOTD to disk based algorithms, both of which can support large sequences of size up to 4 gigabyte. However, they have data skew problem due to their fixed length prefix partition technique. TRELLIS adopts a variable-length prefix partitioning technique to overcome the data skew problem, however its index representation is limited to DNA sequences due to storing only 5 pointers that are corresponding to the DNA symbols (A, G, C, T) and the terminal symbol "$" in its ST nodes, and its index size is larger than others. TRELLIS' capability is also limited to large sequences of size up to 4 gigabyte. While B2ST (same as DIGEST) is based on the binary ST (BST), it can support much longer sequences than others could handle, however, its index representation is not compact.

Our proposed CBST index representation has the following advantages:

1. All the BST nodes are stored on disk files as array format. This allows efficient tree traversals since locating corresponding child nodes could be done in constant time.

2. It includes a two level index structure – a small size lookup table (LT) and relatively much larger BSTs. The LT is a reference to the disk BST files. During query processing, the LT index is resident in the main memory, while the BSTs files are loaded from disk on demand. This avoids many disk I/Os during query operations due to its good locality of references.

3. CBST saves the depth information to the ST branch nodes and suffix starting position in the leaf directly. This is similar to the HST index representation that saves the depth information in leaf nodes to avoid extra jump traverses on the STs. This allows fast tree traversal during search tasks.

4. CBST is a compact, uncompressed disk based ST index representation. It needs only 9 bytes per ST node and 18 bytes per suffix. It can support sequences of size up to 256 terabyte. This is independent of the alphabet of the input, being DNA or otherwise.

Our CBST index representation and associated algorithms can handle chromosome-scale sequences, genome-scale, and beyond. While it is an extension of the B2ST algorithm, it is superior to it in several aspects. CBST requires less space on disk and in main memory due to its compact representation and the “lightweight” suffix sorting algorithm it uses. Compared to TDD, HST and TRELLIS, our CBST algorithm overcomes the memory bottleneck problem that requires the whole input to be resident in the main memory during index construction. In theory, CBST algorithm can handle any size input under a standard personal computer.

The results of our numerous experiments shows that CBST is an efficient and scalable disk based ST technique. It is also the fastest disk-based ST index so far.

During our experiments, we noted that sorting the partition pairs takes most of the index construction time, especially for large sequences. Since sorting partitions could be done independently, paralleizing this phase would result in significant speedup of the whole index construction process. Figure 29 below shows an architecture for this parallelization.

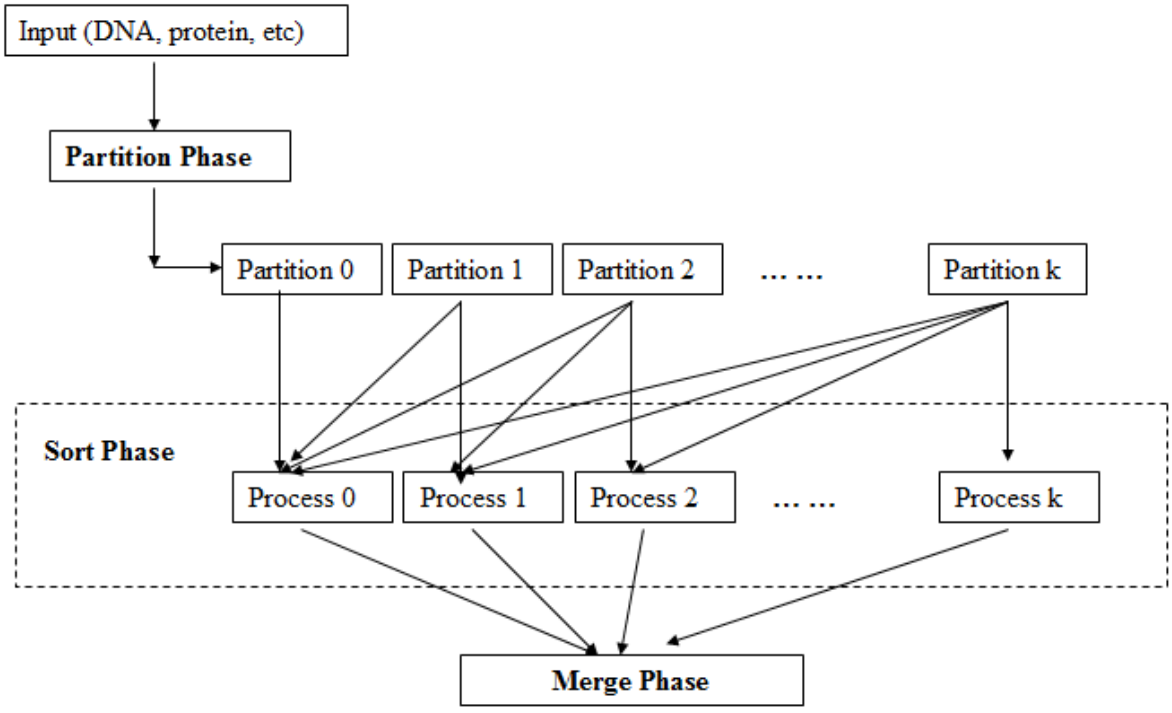


Figure 29. Parallelization of Sorting Partitions

Finally although we considered DNA sequences in our experiments, CBST can be applied to sequences over any alphabet. For example for protein sequences, we only need 5 bits to represent the amino acids alphabet of size 23, instead of 2 bits used for

DNA sequences. Another direction to extend the EM search task proposed here is to develop other search operations based on CBST, like longest repeated substrings (LRS), approximate match, etc.

Bibliography

[1000 Genomes Project, 2011] 1000 Genomes Project, http://en.wikipedia.org/wiki/The_1000_Genomes_Project, last accessed, Jan., 2011

[Abouelhoda et al., 2004] Abouelhoda, M.I., Kurtz, S., and Ohlebusch, E. *Replacing Suffix Trees with Enhanced Suffix Arrays*. In Journal of Discrete Algorithms, Vol. 2(1), pp53-86, 2004

[Apostolico and Galil, 1985] Apostolico, A. and Galil, Z., *The Myriad Virtues of Subword Trees*. In: Combinatorial Algorithms on Words, Vol. 12 of NATO Advance Science Institute Series. Series F: Computer and Systems Sciences. Springer Verlag, Berlin, pp85-95, 1985

[Barsky et al., 2008] Barsky, M., Stege, U., Thomo, A., *A New Method for Indexing Genomes Using On-Disk Suffix Trees*. Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM, pp649–658, 2008.

[Barsky et al., 2009] Barsky, M., Stege, U., Thomo, A., and Upton, C., *Suffix Trees for Very Large Genomic Sequences*. CIKM '09: Proceedings of the 18th ACM Conference on Information and Knowledge Management, 2009

[Barsky, 2011] Barsky, M., Research at UVic, http://webhome.cs.uvic.ca/~mgbarsky/publications.html, last accessed, Jan. 2011

[Bedathur and Haritsa, 2004] Bedathur, S.J., Haritsa, J.R., *Engineering a fast online persistent suffix tree construction*. Proceedings of the 20th International Conference on Data Engineering pp. 720-731, 2004

[Bentley and Mcilroy, 1993] Bentley, J. L. and Mcilroy, M. D., *Engineering a sort function*. Software-Practice and Experience 23, 11, p1249-1265, 1993

[Burrows and Wheeler, 1994] Burrows, M. and Wheeler, D. J., *A block sorting lossless data compression algorithms*. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, CA, 1994

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

[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), pp90-105, 2005

[Dementiev et al., 2005] Dementiev, R., Karkkainen, J., Mehnert, J., and Sanders,P. *Better external memory suffix array construction*. Proc. Of Algorithm Engineering and Experiments, ALENEX'05, pp86-97, 2005

[EST, 2011] Expressed sequence tag, http://en.wikipedia.org/wiki/ Expressed_sequence_tag, last accessed, Jan., 2011

[Farach and Muthukrishnan, 1996] Farach, M., and Muthukrishnan, S., *Optimal Logarithmic Time Randomized Suffix Tree Construction*. Proceedings of the 23rd international Colloquium on Automata, Languages and Programming, LNCS, 1099, pp550-561, 1996.

[Farach et al., 2000] Farach, M., Ferragina, P., and Muthukrishnan, S., *On the sorting complexity of suffix tree construction*. Journal of the ACM, 47 (6), pp987-1011, 2000

[Garcia-Molina et al., 1999] Garcia-Molina, H., Ullman, J. D., Widon J. D., *Database*

System Implementation. Prentice-Hall inc., 1999

[GenBank, 2011] GenBank, http://en.wikipedia.org/wiki/GenBank, last accessed, March, 2011

[Giegerich and Kurtz, 1997] Giegerich, R. and Kurtz, S., *From Ukkonen to McCreight and Weiner: A Unifying View of Linear-time Suffix Tree Construtcion*. Algorithmica, 19(3), pp331-353, 1997

[Giegerich, et al., 2003] Giegerich, R., Kurtz, S., and Stoye, J., *Efficient implementation of lazy suffix trees*. Software Practice & Experience, 33(11), pp1035–1049, 2003.

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

[Gusfield, 2004] Gusfield, D., *Introduction to the IEEE/ACM Transactions on Computational Biology and Bioinformatics*, IEEE Transactions on Computational Biology and Bioinformatics, Vol. 1, No. 1, Jan.-Mar., 2004

[Halachev et al., 2005] Halachev, M., Shiri, N., A. Thamildurai, *Exact Match Search in Sequence Data Using Suffix Trees*. The ACM Conference on Information and Knowledge Management, CIKM'05, Oct. 31-Nov.5, 2005

[Halachev et al., 2007] Halachev, M., Shiri, N., A. Thamildurai, *Efficient and Scalable Indexing Techniques for Biological Sequence Data*, Bioinformatics Research and Development, BIRD'07, pp464-479, March 2007

[Halachev, 2009] Halachev, M., *Management of Biological Sequences Using Suffix*

Trees. A Thesis In the Department of Engineering and Computer Science (ENCS), Concordia, May 2009

[HGP, 2011] Human Genome Project, http://en.wikipedia.org/ wiki/Human_Genome_Projec, last accessed, Jan., 2011

[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, pp256-271, 2002

[Irving and Love, 2003] Irving, R.W. and Love, L., *The Suffix Binary Search Tree a Suffix AVL Tree*, In Journal of Discrete Algorithms, 1 (2003) pp387–408, 2003.

[Larsson and Sdakane, 1999] Larsson N. J., Sadakane K. *Faster Suffix Sorting*, Tech. Rep. LUCS-TR: 99-214, Computer Science Department, Lund University, Sweden, 1999

[Dayhoff, 1965] Dayhoff, M., O., *Atlas of Protein Sequence and Structure*, National Biomedical Research Foundation, 1965

[McCreight, 1976] McCreight, E.M., *A Space-economical Suffix Tree Construction Algorithm*, Journal of ACM, 23 (2), pp262-272, 1976

[Michael and Simon, 2008] Michael A. M., Simon J. P., *An Efficient, Versatile Approach to Suffix Sorting*, Journal of Experimental Algorithmics (JEA), Vol 12, June 2008

[Msufsort, 2011] The Msufsort Algorithm, http://www.michael-maniscalco.com /msufsort.htm, Jan. Last accessed, 2011

[NCBI, 2011] NCBI Genomic Biology, Human Genome Resources,

http://www.ncbi.nlm.nih.gov/genome/gui-de/human/index.shtml, last accessed, 2011

[Phoophakdee and Zaki, 2007] Phoophakdee B., and Zaki, M. J., *Genome-scale Disk-based Suffix Tree Indexing*, ACM International Conference on Management of Data, 2007.

[Phoophakdee and Zaki, 2008] Phoophakdee B., and Zaki M. J., *TRELLIS+: An Effective Approach for Indexing Massive Sequence*, Pacific Symposium on Biocomputing, 2008.

[SACA_Benchmarks, 2011] The benchmark results of implementations of various, latest suffix array construction algorithms, http://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks, last accessed, 2011

[Seward, 2011] Seward, J., The bzip2 and libbzip2 homepage, http://sources.redhat.com/bzip2/, Mar., 2011

[Simon, 2005] Simon J. Puglisi, *Exposition and Analysis of a Suffix Sorting Algorithm*, Technical Report Number CAS-05-02-WS, Dept of Computing and Software, McMaster University, May, 2005

[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

[Thamildurai, 2007] Thamildurai, A., *Efficient and Scalable Indexing Techniques for Sequence Data Management*. A Thesis In the Department of Engineering and Computer Science (ENCS), Concordia, April 2007

[Tian et al., 2005] Tian, Y., Tata, H., Hankins, R., Patel, J., *Practical methods for*

constructing suffix trees. The VLDB Journal, 14(3), pp281–299, 2005.

[Ukkonen, 1995] Ukkonen, E., *On-line Construction of Suffix Trees*. Algorithmica, 14 (3), 1995

[USCS, 2011] USCS Genome Browser, hgdownload.cse.ucsc.edu /downloads.html, last accessed 2011

[Vmatch, 2011] The Vmatch large scale sequence analysis software, http://www.vmatch.de/, last accessed, Jan. 2011

[Weiner, 1973] Weiner, P., *Linear pattern matching algorithms*. Proc. 14th Annual Symposium on Switching and Automata Theory, 1973

[Wikipedia, 2011] Suffix tree, http://en.wikipedia.org/wiki/Suffix_tree, last accessed July, 2011