TRELLIS: GENOME-SCALE DISK-BASED SUFFIX TREE INDEXING ALGORITHM

By

Benjarath Phoophakdee

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Prof. Mohammed J. Zaki, Thesis Adviser

Prof. Christopher Bystroff, Member

Prof. Lee Newberg, Member

Prof. David Spooner, Member

Prof. Bülent Yener, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2007
(For Graduation August 2007)

TRELLIS: GENOME-SCALE DISK-BASED SUFFIX TREE INDEXING ALGORITHM

By

Benjarath Phoophakdee

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Prof. Mohammed J. Zaki, Thesis Adviser
Prof. Christopher Bystroff, Member
Prof. Lee Newberg, Member
Prof. David Spooner, Member
Prof. Bülent Yener, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2007
(For Graduation August 2007)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Mohammed J. Zaki, who gave me a chance to pursue a PhD degree. He is the first professor I had a class with at RPI and has been my advisor for seven years. Without his continual support, finish- ing a PhD would be impossible. I thank him for teaching me how to do research, for all lessons learned, valuable advices, and his tireless patience. I also would like to thank Dr. Christopher Bystroff, Dr. Lee Newberg, Dr. David Spooner, and Dr. B?ulent Yener for agreeing to be on my thesis committee and for their valuable feedback.

Next I would like to thank my parents for their unimaginable amount of love, support, patience, and everything they have done for me. I am eternally grateful for such happy and fulfilled childhood, which provides a strong ground for the person that I am today. Without them, none of this would be possible. Many thanks are also due to my eldest sister, Benjamas, who is like my second mother. Thank you for always looking over me. I also would like to thank my two other siblings, Anusha and Anchalee, for always being there to help me, regardless. I also must thank my little sister, Benjawan, for showing me how not to lose my sense of humor during this challenging endeavor.

My life in the United States would not have been as enjoyable without my best friend, Dr. Oratai Jongprateep. She has been a constant source of advice, love, and support for more than a decade. I could not have wished for a better friend, and for that I thank her.

I would like to thank Asif Javed for his help, support, and understanding. Thank you for reading my manuscripts and more importantly for providing strength and calmness whenever I needed them. Next I would like to thank my well-behaved dog, Mr. Ninja Momotaro, whom I could not be prouder of. Thank you for teaching me patience, responsibilities, and time management, and for being a stable source of happiness for the past six years. I owe greatly to my dog.

Finally, I would like to thank God Almighty who planted the desire in my heart twenty-six years ago. There I was on the steps in front of my kindergarten classroom, counting my fingers to see how many weeks it would take until I finish my PhD. Little did I know, and now I know. Thank you for all the blessings in my life.

ABSTRACT

With the exponential growth of biological sequence databases, it has become critical to develop effective techniques for storing, querying, and analyzing these massive data. Suffix trees are widely used to solve many sequence-based problems, and they can be built in linear time and space, provided the resulting tree fits in main-memory. To index larger sequences, several external suffix tree algorithms have been proposed in recent years. However, they suffer from several problems such as susceptibility to data skew, non-scalability to genome-scale sequences, and non-existence of suffix links, which are crucial in various suffix tree based algorithms.

In this thesis, we propose a novel disk-based suffix tree algorithm for indexing DNA sequences called Trellis. Our algorithm does not suffer from any of the above drawbacks, effectively scales up to genome-scale sequences, and is also able to quickly rebuild suffix links. Specifically it can index, on a typical modern computer, the entire human genome using 2GB of memory in about 4 hours and can recover all the suffix links within an additional 2 hours. Despite the success of Trellis, the algorithms main limitation is that it requires the entire input sequence to be kept in memory. To handle larger DNA sequences, we introduce a novel string buffering strategy that allows our algorithm to assign a very small amount of memory for the input string. As a result, Trellis is able to index very large sequences in a reasonable amount of time and memory. The buffer strategy also speeds up Trellis when the input string barely fits in memory. Trellis was compared to various state-of-the-art persistent disk-based suffix tree construction algorithms, and was shown to outperform the best previous methods, both in terms of indexing time and querying time.

CHAPTER 1
INTRODUCTION

Over the past years, DNA sequence databases have been growing at exponential rates due to the various sequencing efforts throughout the world. Recently the GenBank [31] DNA sequence database has crossed the 100 Gbp[1] mark, with sequences from over 165,000 organisms. As a consequence of the enormous data size and extreme growth rate, it has become critical for researchers to have effective data structures and efficient algorithms for storing, querying, and analyzing these sequence data.

A suffix tree is a versatile data structure that can be used to solve a variety of sequence-based problems, such as exact and approximate matching, database querying, finding the longest common substrings, and so on [22]. Suffix trees have also been extensively used in genome alignment approaches to find matching segments [12, 4, 26].

Suffix trees can be constructed in linear (in the sequence length) time and space [30, 38, 36], provided the tree fits entirely in the main memory. A variety of efficient in-memory suffix tree construction algorithms have been proposed [20, 17, 16, 37, 15]. However, these algorithms do not scale up when the input sequence is extremely large (for example, the human and mouse genomes are approximately 3 Gbp and 2.5 Gbp long, respectively). This is mainly because the random memory accesses to the input sequence as well as the suffix tree, during construction, result in poor locality of reference and memory bottlenecks [17, 27, 35].

Several disk-based suffix tree algorithms have been proposed recently. Some of the approaches [27, 28, 33, 34, 35] completely abandon the use of suffix links and sacrifice the theoretically superior linear construction time in exchange for a quadratic time algorithm with better locality of reference. However, many fast string-processing algorithms, such as tandem repeats [23], structural motifs [7], ap-

[1]We report sequence length in the number of *base pairs* (bp); K, M, and G stand for 10^3, 10^6, 10^9, respectively; a DNA base is one of A, C, G or T, and they occur in complementary pairs in a double stranded DNA molecule.

proximate string matching [8], and genome alignment [12, 4, 26], rely heavily on suffix links for efficiency, and thus cannot be used directly with these disk-based suffix trees. Some approaches [27, 28, 33, 5] also suffer from the *skewed partitions* problem. They build prefix-based partitions of the suffix tree relying on a uniform distribution of prefixes, which is generally not true for sequences in nature. This results in partitions of non-uniform size, where some are very small, and others are too large to fit in memory. Methods that do not have the skew problem and that also maintain suffix links, have also been proposed [3, 9]. However, these methods do not scale up to the human genome level.

1.1 Thesis Contribution

In this work, we present TRELLIS[2], a novel algorithm to construct genome-scale suffix trees on disk. Specifically, we make the following contributions:

1. TRELLIS is an $O(n^2)$ time and $O(n)$ space, disk-based suffix tree construction algorithm (where n is the sequence length) based on the idea of constructing the tree by *partitioning and merging*. It uses *variable-length prefixes* to solve the partition skew problem afflicting several previous disk-based algorithms. TRELLIS has a fast post-construction phase to *recover the suffix links* for the complete suffix tree.

2. TRELLIS scales gracefully for very large DNA sequences. Using only 2GB of memory, it can index the entire human genome in about 4 hours and can recover the full set of suffix links within an additional 2 hours. To the best of our knowledge TRELLIS is the first disk-based suffix tree algorithm *with suffix links* that is capable of indexing the human genome with a limited amount of memory.

3. TRELLIS outperforms the current state-of-the-art algorithms that construct external suffix trees (both with and without suffix links). Specifically, TRELLIS was shown to be faster than the leading algorithm that constructs the

[2]TRELLIS is 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

human genome suffix tree (without the suffix links) [35] by a factor of 2-4 times. We also compared TRELLIS with the only algorithms that do not exhibit the data skew problem and that also maintain the suffix links [3, 9]. Our experiments indicate that these algorithms can only handle chromosome-scale input sequences and TRELLIS outperforms these methods by several orders of magnitude.

4. We also study the query-time performance over genome-scale suffix trees. More specifically, we ran query experiments over a disk-based suffix tree for the entire human genome. We show that TRELLIS query times can be between 2-15 times faster than existing methods; each query takes on average between 0.01-0.06s, depending on whether or not suffix links are used. To the best of our knowledge this is the first paper to report query times over the disk-based suffix tree for the entire human genome.

5. We introduce TRELLIS-2 a faster and more scalable version of TRELLIS. The improvements are due to the two optimization techniques developed. The first optimization allows faster suffix tree construction. Specifically, TRELLIS-2 was shown to be about two times faster than its original counterpart. The second optimization allows TRELLIS-2 to index sequences much larger than the size of the memory. Our algorithm was able to index the entire human genome in about 11 hours, using 512MB of memory .

1.2 Thesis Outline

We begin by discussing the preliminaries and related work in Chapter 2 and 3. The TRELLIS algorithm is introduced in Chapter 4. TRELLIS-2, the optimized version of TRELLIS, is described in Chapter 5. Chapter 6 presents experimental results and discussion. Finally, Chapter 7 summarizes the main contributions of this thesis and suggests possibilities for future work.

CHAPTER 2
PRELIMINARIES

Let Σ denote a set of characters (or the alphabet), and let $|\Sigma|$ denote its cardinality. Let $\Sigma^\star$ be the set of all possible strings (or sequences) that can be constructed using Σ. Let $\$ \notin \Sigma$ be the *terminal* character, used to mark the end of a string. Let $S = s_0 s_1 s_2 \ldots s_{n-1}$ be the input string where $S \in \Sigma^\star$ and its length $|S| = n$. The i^{th} suffix of S is represented as S_i, with $S_i = s_i s_{i+1} s_{i+2} \ldots s_{n-1}$. For convenience, we append the terminal character to the string, and refer to it by s_n.

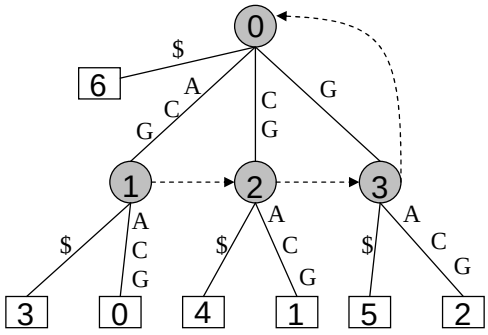


Figure 2.1: Suffix tree T_S for string $S =$ ACGACG\$. The circles represent internal nodes, and the squares represent leaf nodes. The leaf nodes are numbered with respect to their corresponding suffixes. The internal nodes are numbered with respect to depth-first ordering.

The suffix tree of the string S, denoted as T, stores all the suffixes of S in a tree structure, where suffixes that share a common prefix lie on the same path from the root of the tree. A suffix tree has two kinds of nodes: internal and leaf nodes. An internal node in the suffix tree, except the root, has at least 2 children, where each edge to a child begins with a different character. Since the terminal character is unique, there are as many leaves in the suffix tree as there are suffixes, namely $n + 1$ leaves (counting \$ as the "empty" suffix). Each leaf node thus corresponds to a unique suffix S_i and is denoted as L_i. Each node (internal or leaf), v, is associated

with its depth, $d(v)$, which is equal to the number of the edges on the path from the root to v. Let $\sigma(v)$, called the label of v, denote the substring obtained by traversing from the root to v and concatenating all the characters on that path. Every internal node v, with $\sigma(v) = x\alpha$ (where $x \in \Sigma$ and $\alpha \in \Sigma^\star$), has a *suffix link*, $sl(v) = w$, that points to an internal node w such that $\sigma(w) = \alpha$. A suffix tree T for an example DNA sequence, $S = ACGACG\$$ is shown in Figure 2.1. Here the dashed arrows indicate the suffix links. For example, there is a suffix link from the node with label ACG to one with label CG.

A straightforward approach to constructing suffix trees works in $O(n^2)$ time and space. The method simply inserts each suffix S_i (of length $n-i+1$) into the tree starting from the root. Since there are n suffixes, and an insertion takes $n-i+1$ time, we get a total time of $O(n^2)$. Also, since there are n leaves, and each path to a leaf is labeled, the tree consumes $O(n^2)$ space. However, as we noted earlier, there exist efficient memory-based suffix tree construction methods that work in linear ($O(n)$) time and space [30, 38, 36], provided the tree fits entirely in the main memory. For example, Ukkonen's [36] algorithm employs several techniques to achieve linear time/space, which include: suffix links, edge encoding, and skip-and-count. Suffix links are pointers from any internal node v to its suffix $sl(v)$, as shown with dashed arrows in Figure 2.1 and 2.2.

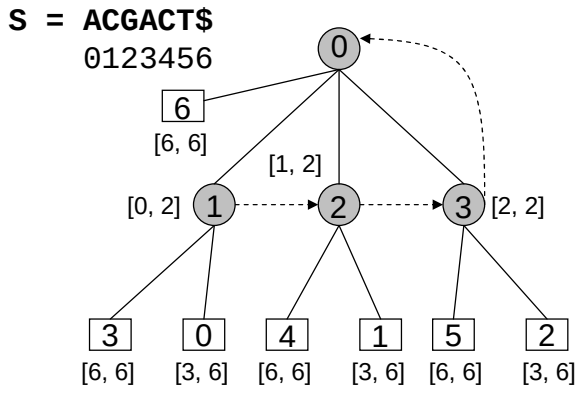


Figure 2.2: Suffix tree T_S for string S = ACGACG$ using edge index encoding. The encoding enables the suffix tree to require only $O(n)$ space instead of $O(n^2)$.

Edge encoding is used to reduce space to $O(n)$. The idea is to replace each

edge with the start and end indexes (positions) in the input string S for the label (substring) on the edge. For example, the edge labeled ACG from the root node, will be encoded as $[0, 2]$, whereas edge labeled ACG leading to leaf (suffix) 0, will be encoded as $[3, 6]$. Figure 2.2 illustrates the use of edge index encoding. Using edge-encoding, each edge requires only a constant amount of space, so the total space required for the whole tree is $O(n)$.

Let $X = y\alpha\beta$ and let $Y = \alpha\beta z$ be any two strings, with $y, z \in \Sigma$ and $\alpha, \beta \in \Sigma^*$. Skip-and-count means that if we search the suffix tree for X followed by Y, and if we have already matched X up to $y\alpha$, and there is a mismatch at the first character of β, then we can search for Y from the root by skipping over intermediate nodes and counting $|\alpha|$ characters. Alternately, we can use the lowest suffix link on the path $y\alpha$, jump to the suffix node and skip-and-count the remaining matching characters, before continuing to search for the characters in β. The intuition behind the skipping and counting technique is that, for any string X found in a suffix tree, any of X's suffixes must also be found in the suffix tree. Suffix links provide a quick route to reach such suffixes. There is no need to match them character by character, since we already know that they must be present in the tree. For example, let suppose we first search for the string $ACGAC$ in the suffix tree above. We find that it ends at the leaf denoting the 0^{th} suffix. Next let suppose we need to search X's 1^{st} suffix, $CGAC$. There is no need to start searching now from the root. We can simply go up one step to the internal node 1, follow the suffix link to the internal node 2, where we know that we have matched up to the second character of $CGAC$, then skip-and-count down for two more characters to find the rest of $CGAC$. The simultaneous use of these three techniques (suffix links, edge encoding, and skip-and-count) results in linear time and space suffix tree construction.

CHAPTER 3
RELATED WORK

Hunt et al. [27] introduced one of the first disk-based suffix tree construction algorithms. They studied Kurtz's method [29], which at the time acquired the most memory efficient suffix tree representation (about 13 bytes per character indexed). Kurtz's method, like many other fast in-memory suffix tree construction algorithms, relies heavily on suffix links to achieve its fast linear time tree construction. However, Hunt et al. found that Kurtz's approach cannot be ported immediately to a disk-based environment due to its poor locality of tree node reference caused by suffix links. Simply put, these algorithms must access the nodes deep down the tree as well as across the tree via suffix links during the construction. Such behavior was beautifully described by Giegerich and Kurtz [21] as follows: "*The active suffix creeps through the text like a caterpillar. At the same time, the corresponding active node swings through the tree like a butterfly*"

In order to gain a better locality of reference, Hunt's approach abandons the use of suffix links and sacrifices the theoretically superior construction time of $O(n)$. The method first calculates a set of *fixed*-length prefixes of the input string, such that the suffix sub-tree from strings having a given prefix, can fit entirely in main memory. The static pre-partitioning method via fixed-length prefixes assumes that the DNA sequence has uniform base distribution. For each fixed-length prefix, the method makes a pass over the input string and inserts all suffixes starting with the given prefix into a disk-based suffix, therefore requiring multiple passes over the input string in total. Hunt's method has $O(n^2)$ complexity in the worst case. They claimed that, assuming the uniformly distributed nature of DNA, their method is $O(n \log n)$ in practice.

However, the characters in real DNA sequences are not uniformly distributed. Consequently, some partitions may fit in the memory while some may not. We gathered some statistics from the human genome and displayed the frequencies of length-2 and length-3 prefixes in Figure 3.1 below. It is very obvious from the

graphs that fixed-length prefixes will surely result in largely unequal partition sizes. As a result, Hunt's method exhibits difficulty in handling data skew. The authors suggest the use of bin-packing to solve this data skew problem, however, frequency counting for long fixed-length prefixes is required prior to bin-packing, which can be very expensive, since there are 4^l possible prefixes of length l for DNA sequences.

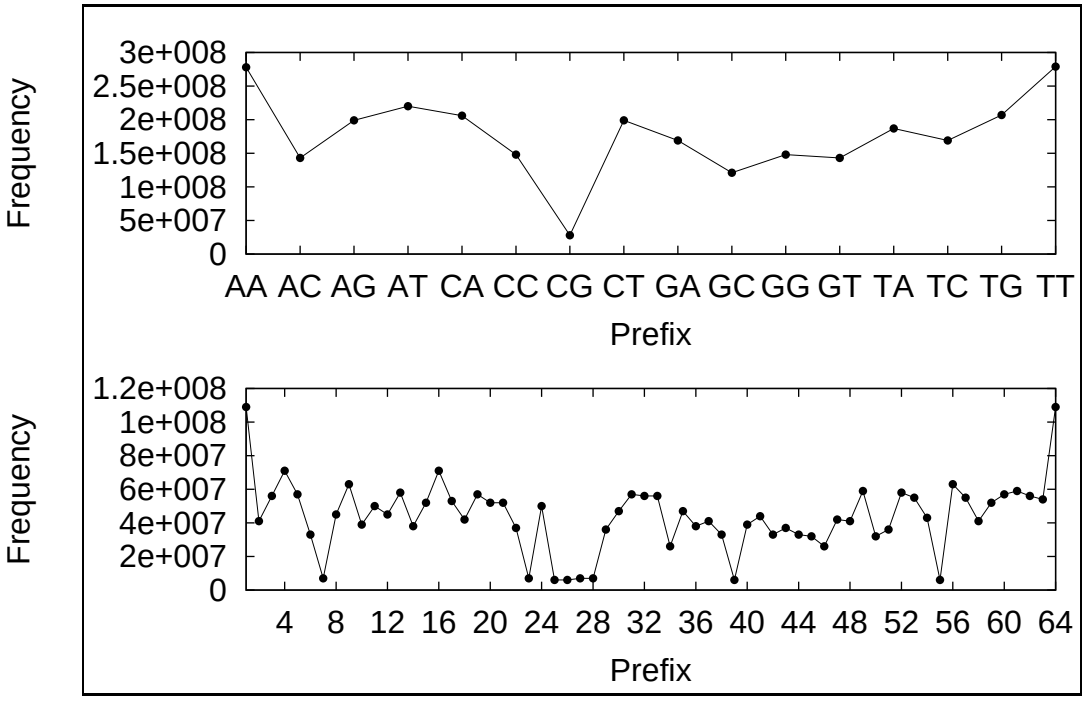


Figure 3.1: Frequency of length-2 and length-3 prefixes in the human genome. There are 16 and 64 prefixes respectively. For length-3 prefixes, the data points represent prefix AAA, AAC, AAG, ..., TTT, in that order.

The suffix link disposal in Hunt's method also affects the suffix tree size on disk. Typically, the number of internal nodes in a suffix tree is 0.6 – 0.8 times the number of leaf nodes (which is n). Their method arguably saves on space to store suffix links in that order. However, this means their disk-based suffix trees cannot be used by various suffix tree-based algorithms, such as approximate string matching [8] and genome alignment [12, 4, 26], whose fast performance relies on the existence of suffix links.

In [33], the authors improved upon Hunt's algorithm by storing the subtrees, i.e., partitions, separately in clusters instead of all together in one suffix tree. The

main difference between this method and Hunt's is that insertions of new suffixes do not start at the tree's root, but rather at the depth of the fixed-length prefix. The authors state that their algorithm is suitable for the construction of large suffix trees as long as the memory size is six times bigger than the input sequence. This means, for 2GB of memory, which is not uncommon on a modern computer, this algorithm is suitable for indexing sequences only up to about 340Mbp in length.

Top-compressed suffix trees were introduced in [28]. The author improved upon Hunt's algorithm by introducing a pre-processing stage and by using parallel index construction. The insertion of suffixes into partitioned suffix trees is also optimized via the use of suffix links. Although the algorithm maintains the suffix link structure, it assumes that the data can be equally partitioned and therefore suffers from the data skew problem.

Cheung et al. [9] described three major drawbacks in Hunt's approach: large I/O overhead due to tree traversal during the suffix tree construction, substantial random disk accesses with no locality, and difficulty in handling data skew. The authors addressed all three problems via dynamic clustering. In DynaCluster [9], the suffix tree is created one cluster at a time in a top-down manner. A cluster is a group of nodes that are likely to be referred to together during the suffix tree construction. Each cluster requires only a few disk pages. Therefore, the clusters provide locality which allows a large disk-based suffix tree to be built using a limited size of memory. A main difference between DynaCluster and other external disk-based suffix tree algorithms evaluated in this thesis is that DynaCluster is insensitive to the amount of memory available. Since the tree is created by only focusing on one small cluster at a time, the algorithm does not assume any data distribution and therefore can handle any data skew. In comparison to Hunt's approach, DynaCluster can reduce both CPU and I/O cost. DynaCluster also drops the use of suffix links during the tree construction to allow a better locality of reference; thus its theoretical complexity is $O(n^2)$. However, similar to most disk-based suffix tree methods with $O(n^2)$ complexity, its experimental behavior is actually $O(n \log n)$.

DynaCluster is the first algorithm that provides an optional phase that rebuilds the suffix links after the entire tree is constructed on disk. Their suffix link rebuild

method requires three additional large data structures, each requiring $O(n)$ of disk space. The first data structure is the least common ancestor table (*lca-table*) in which the information needed for answering lca queries is maintained. The lca-table is reported to require 12 bytes per character indexed. The second data structure is called the *leaf-table*, which stores pointers to leaf nodes in the tree. The last data structure is call the *two-leaf table* which stores additional information to aid in lca lookups. The disk-based tree needs to be traversed twice during the link recovery. Prior to restoring a suffix link to an internal node, the algorithm issues look-ups in all of the above additional data structures. Such disk access patterns exhibit poor locality, i.e. the disk head must traverse the disk-based tree as well as seek through the three data structures per internal node before a suffix link can be added. Such poor locality is reflected in their experimental data where the suffix link rebuild time is reported to be much slower than its tree construction time.

Bedathur et al. [3] developed the TOP-Q buffer management policy of suffix tree nodes for online disk-based suffix tree construction. Online disk-based suffix tree construction was previously considered impractical due to its excessive disk I/O cost, but the authors were able to demonstrate that a careful choice of implementation strategy can make it possible in practice. TOP-Q is built upon the linear time Ukkonen [36] algorithm and thus results in a disk-based suffix tree construction algorithm that does not sacrifice the suffix links. TOP-Q focuses on indexing DNA and Protein sequences. A highlight of the algorithm is that it does not assume any specific data distribution which makes it suitable for biological sequences in nature. TOP-Q buffer management policy was designed based on the observation that nodes closer to the top of the tree are likely to be accessed more during the online tree construction. Due to their higher node access frequency, those nodes should be retained in the buffer. The authors showed that the node's path length (the total number of characters from the root to that node) provides an upper-bound of the node's eventual depth in the tree, and thus can be use as an indicator of whether to retain this node in memory. Tree nodes are packed together to form a buffer page. A buffer page either stays or is evicted from the memory based on the average node path length of all the nodes in that page. TOP-Q additionally

maintains a short and fixed-length queue to store nodes recently evicted from the buffer. Their experimental results shown that the queue introduces a delay in the eviction of buffer pages and effectively reduces the buffer miss rate which leads the algorithm to perform better in practice. Another interesting aspect of this work is the study of tree node representation. The authors compared the difference between storing children nodes in a linked-list structure versus storing them in a fixed-size array and found that the former requires far less space than the latter. However, the linked-list representation incurs a significantly higher I/O overhead than the array representation. Therefore, array seems to be a better choice for storing children nodes in a disk-based suffix tree.

TOP-Q and DynaCluster are currently the only algorithms that maintain suffix links as well as do not exhibit the data skew problem. However, neither of them has been reported experimentally to scale up to the human genome level. Our experiments also indicate that they do not scale beyond the chromosome level (on the order of ≈ 100 Mbp). The authors of TOP-Q recently also developed an effective layout scheme called Stellar [2] to support queries over disk-based trees.

A disk-based $O(n^2)$ suffix tree construction method, called TDD, was described in [34, 35]. Similar to Hunt's approach, TDD drops the use of suffix links in exchange for a much better locality of reference. It uses a Partition and Write Only Top Down suffix tree method, which is based on the wotd-eager algorithm [20], combined with a specialized buffering strategy that allows better cache usage during tree construction. TDD was experimentally shown to outperform other disk-based suffix tree algorithms. Its superior performance is due largely to the use of specialized buffers which are *input string buffer*, *suffix tree node buffer*, *suffixes buffer*, and *temp buffer*. The last two buffers are used for managing the list of suffixes to be inserted into the tree. The main idea behind TDD's buffering scheme is in choosing appropriate buffer sizes and page replacement policies. Similar to most external suffix tree algorithms, the input string exhibits poor locality of reference due to random accesses incurred during the tree construction. Therefore, it receives the highest priority in buffer space assignment. The page replacement policy used for the *string buffer* is the least recently used (LRU) policy. The *tree buffer* requires the

largest disk space among all buffers, however it has very good temporal and spatial locality because it consists mainly of sequential writes with occasional node revisits. Therefore, it is assigned the least buffering space in memory with LRU replacement policy. The last two buffers are used for suffix sorting and are of equally large size. The *suffixes buffer* requires sequential scans with some almost random writes, thus LRU policy seems suitable. The *temp buffer* requires exactly two linear scans, therefore the most recently used (MRU) page replacement policy works best for it.

Prior to our work [32], TDD was the only algorithm reported to scale up to the human genome level. The authors discovered that the main disadvantage of TDD is the random I/O incurred when the input string cannot reside entirely in the main memory. As an improvement to TDD, the ST-Merge method was introduced in [35]. It separates the input string into smaller contiguous substrings, applies TDD to build a suffix tree for each substring, and later merges all the trees together into a complete suffix tree. ST-Merge was experimentally shown to outperform TDD when the input string is several times larger than the main memory, however their experiments were only reported up to the chromosome level; ST-Merge has not been shown experimentally to index the entire human genome. Both TDD and ST-Merge also suffer the same drawback which is the lack of suffix links.

Lastly, we would like to note that paged and distributed suffix trees were proposed in [10]. They introduce an approach based on *sparse* suffix trees, which are subtrees of the whole suffix tree. These sparse trees have only internal suffix links, and links across the different subtrees are ignored. The method was reported to scale to 200Mbp sequences (chromosome scale). Other suffix trees variants [25], and other disk-based sequence indexing structures like String B-trees [18] and external suffix arrays [13, 11] have also been proposed to handle large sequences.

CHAPTER 4
THE TRELLIS ALGORITHM

We now present our novel disk-based suffix tree construction algorithm. TRELLIS has 4 main steps:

1. *Prefix Creation Phase:* The first step creates a list of variable-length prefixes that ensure that the data skew problem will not occur.

2. *Partitioning Phase:* In the second phase the input string is partitioned, and TRELLIS builds separate prefix-based suffix subtrees from each partition.

3. *Merging Phase:* In this step, TRELLIS merges the suffix subtrees from all partitions into a single suffix tree per variable-length prefix.

4. *Suffix Link Recovery Phase:* The last step is optional; it constructs the entire set of suffix links should they be needed.

We look at each of these phases in detail below. An overview of the algorithm is shown in Figure 4.1.

4.1 Prefix Creation Phase: Computing Variable Length Prefixes

Fixed-length Prefixes: The idea of partitioning a suffix tree based on the fixed-length prefixes of each suffix was originally introduced in [27]. However, it has difficulty in handling data skew due to the fact that DNA sequences do not have a uniform base distribution. For example, the frequencies of length-1 prefixes in the human genome, i.e., A, C, G, and T, are about 30%, 20%, 20%, and 30%, respectively. Applying the fixed-length prefix technique to real DNA data typically results in a great portion of partitions being larger than expected [9]. In addition, computing the appropriate prefix length l is not described in any of the previous works. Large values of l may result in too many partitions, with several partitions being smaller than necessary and wasting resources. Small values may cause some

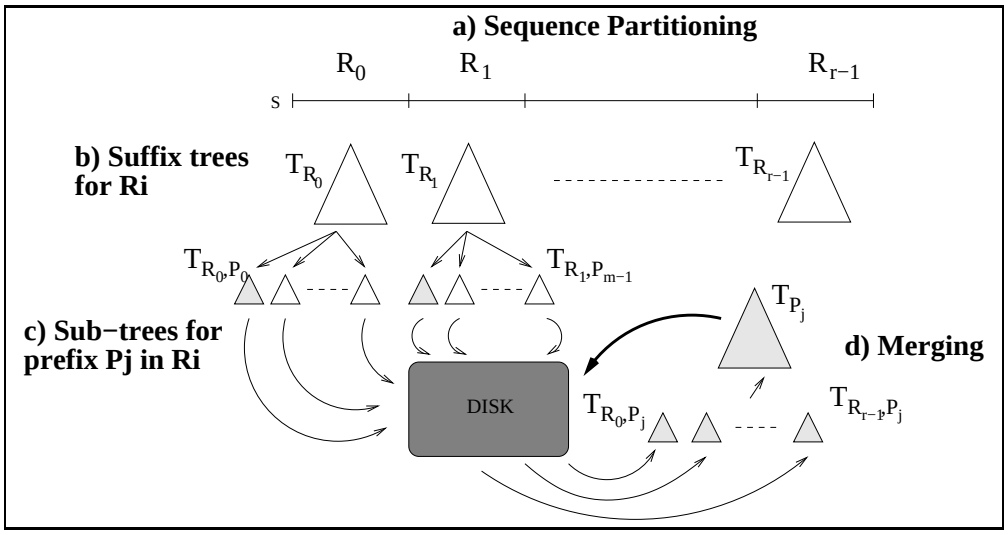


Figure 4.1: Overview of TRELLIS: Given input sequence S, we first partition it into segments R_i (step a). The segment size is chosen (based on a threshold t) such that the resulting suffix tree T_{R_i} from each segment (step b) fits in main memory. Each resulting suffix tree is further split into smaller subtrees T_{R_i,P_j} (step c), that share a common prefix P_j, which are then stored on the disk. After processing all partitions R_i, we merge all the subtrees T_{R_i,P_j} for each prefix P_j from the different partitions R_i into a merged suffix subtree T_{P_j} (step d). The prefixes P_j are chosen based on a threshold t, hence their suffix subtrees also fit entirely in memory. As each merged subtree T_{P_j} is constructed, it is written to disk. The complete suffix tree is simply a forest of these prefix-based subtrees (T_{P_j}).

partitions to be larger than the memory, requiring tree node buffering, and thus incurring additional disk I/O cost. To avoid the data skew problem, a bin-packing technique was suggested in [27], but not implemented.

Variable-length Prefixes: Instead of using fixed-length prefixes, TRELLIS uses variable-length prefixes, based on the observation that not all prefixes need to be extended to a longer length for their partition to fit in memory.

Let $P = \{P_0, P_1, P_2, \ldots, P_{m-1}\}$ denote the set of *variable-length prefixes* of S. Let *freq(P_i)* denote the frequency of the prefix P_i occurring in S. The threshold t is used to determine P, such that $freq(P_i) \leq t$ for all $i \in [0, m)$, i.e., we require that all prefixes in P have frequency at most t. Note that the threshold t is chosen based on the amount of memory available for tree construction (see Section 4.5). Prefix

frequency statistics from the human genome are shown in Table 4.1. For simplicity, we set the value of t to 10^6. We observed that the number of prefixes with frequency more than t sharply decreases as the prefix length increases. In fact, there are only two such prefixes starting at length eight (contrast this with the fact that there are potentially 4^l prefixes of length l).

Table 4.1: The number of prefixes with frequency more than $t = 10^6$. Starting from length 8, there are only 2 such prefixes. At length 16, there are no remaining prefixes with frequency more than t.

Length	#Prefixes	Length	#Prefixes
1	4	5	781
2	16	6	930
3	64	7	68
4	253	8-15	2

We adopt a multi-scan approach to efficiently compute all variable-length prefixes. During each scan, we process the prefixes up to a certain length, such that the data structures needed for frequency counting will fit in memory. Let's assume that the maximum number of prefixes that can be counted in any stage is also bounded by the threshold t. Let L_i denote the longest prefix length after the i^{th} scan, and let EP_i denote the set of prefixes that require further extension in the next scan (i.e., those prefixes having frequency $> t$). At each stage we set L_i, such that $|EP_i| \sum_{j=1}^{L_i - L_{i-1}} |\Sigma|^j \leq t$, where $|EP_0| = 1$ and $L_0 = 0$. At each stage we add to P only the smallest length prefixes that meet the frequency threshold t, and reject all their extensions. For example for the human genome, with $t = 10^6$, only two stages were required, with $L_1 = 8$ and $L_2 = 16$. As per Table 4.1, only two prefixes of length 8 have frequency $> t$, and at length 16, none remain (since $4^{16} >$ 3Gbp). The resulting set P of variable-length prefixes for the human genome contains prefixes ranging from length 4 to 16, and there are 6400 variable-length prefixes obtained in total.

4.2 Partitioning Phase: Creating Prefixed Suffix Subtrees

TRELLIS divides the input string S into $r = \lceil \frac{n+1}{t} \rceil$ consecutive substrings or

partitions. Let R_i denote the i^{th} partition and let T_{R_i}, called a *suffix subtree*, be the tree containing all the suffixes of S that start in R_i.

Each partition can be processed entirely in main memory, since by design, each subtree T_{R_i} can have at most t suffixes, and t is chosen such that a suffix tree with t leaves can fit in memory. We adopt Ukkonen's algorithm [36] for creating the subtrees because of its efficient $O(t)$ time/space complexity. After each T_{R_i} is built in the memory, it is separated further into several subtrees prior to being stored on disk. The separation is according to the set P, i.e., we split T_{R_i} into smaller subtrees T_{R_i,P_j}, called *prefixed suffix subtrees*, which consist of only those suffixes (paths) from T_{R_i} that begin with the a given prefix $P_j \in P$. Therefore, for each partition R_i of the input string, at most m files will be created, where each file represents the subtree T_{R_i,P_j} (with $j \in [0, m-1]$, and $i \in [0, r-1]$). See Figure 4.1 for an illustration of this step.

During the partitioning step, suffix trees $T_{R_i}; \forall i, 0 \leq i < r$ are constructed. Directly applying the Ukkonen's algorithm to each partitioned string R_i results in an *implicit* suffix tree, where some of the suffixes are implicitly part of internal edges. However, for the last partition R_{r-1}, we do obtain an *explicit* suffix tree, because the terminal character is added to the end of the input string, which results in all suffixes being explicitly present in the tree. In order to guarantee that T_{R_i} contains explicitly all of the suffixes from the i^{th} partition, we must make sure that T_{R_i} has exactly $|R_i|$, i.e., t leaves.

One way to solve this problem is to add the terminal character \$ at the end of each partitioned string R_i. This will cause T_{R_i} to have $|R_i|$ leaves. However, this approach may cause problems in the merging phase, since \$ is not really supposed to be at that index of the input string. We implemented this approach in an earlier version of TRELLIS, but we found that it incurs additional overhead during the merging phase. Instead of stopping the tree construction when exactly t characters have been read, TRELLIS solves the problem by continuing to read some of the characters from the next partition, R_{i+1}, until t leaves are explicitly obtained in T_{R_i}. We continue to read characters until the first time a unique prefix is encountered, at which point all suffixes of T_{R_i} become explicit. This step does not add any additional

overhead since typically only a few additional characters $c \ll t$ suffice to make the trees explicit. That is, instead of reading t characters, we read $t + c$ characters for each partition R_i, for $i \in [0, r-2]$. For example for the human genome with $t = 10^6$ at most $c = 1000$ additional characters were required to be read.

4.3 Merging Phase: Merging Prefixed Suffix Subtrees

All prefixed suffix subtrees of the same prefix from the previous step are loaded into the memory (since $freq(P_j) < t$ for all $P_j \in P$) and merged together to form a suffix subtree for that prefix, i.e., *a prefixed suffix tree*. The resulting tree is then stored back to the disk. More specifically, the merging for a given prefix P_j proceeds in steps. At each stage i, let M_i denote the current merged tree obtained after processing subtrees T_{R_0,P_j} through $T_{R_i,P,j}$. In the next step we load T_{R_{i+1},P_j} into memory, and merge it with M_i to obtain M_{i+1}, and so on (for $i \in [0, r-1]$). The final merged tree M_{r-1} is the full prefixed suffix tree T_{P_j}, which is then stored back on the disk. The algorithm continues until all variable-length prefixes P_j $(0 \leq j < m)$ are processed.

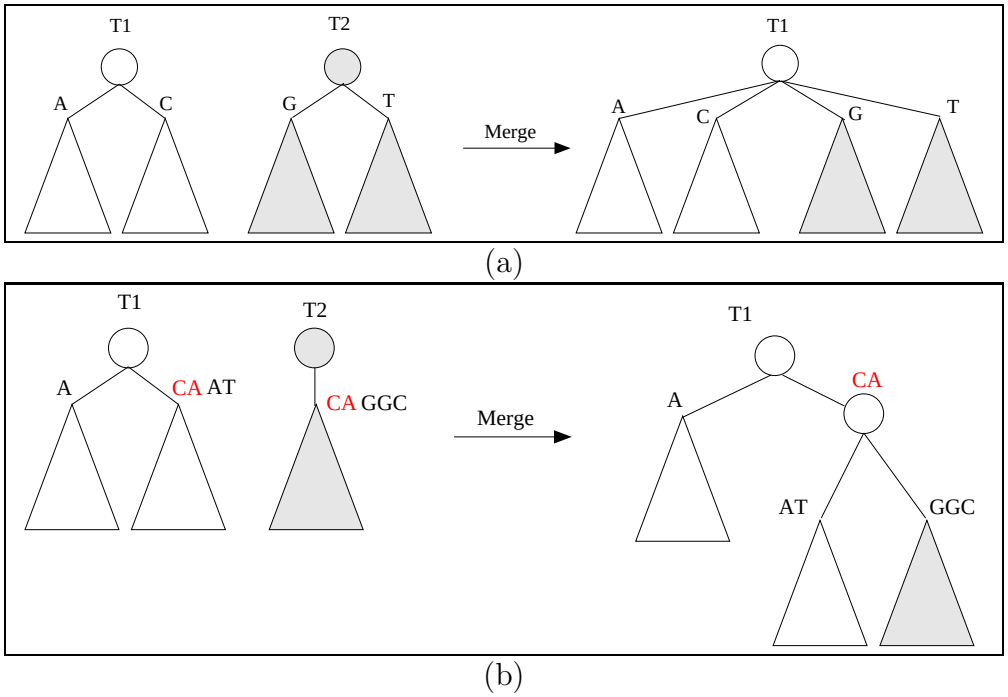


Figure 4.2: Merging Trees: (a) simple case, with disjoint labels. (b) case when some edges have a common prefix.

Given two trees, M_i and T_{R_{i+1},P_j}, the tree merge algorithm recurses from the two root nodes, and for any two nodes v_1 and v_2 in the two trees, that have the same path label, it merges their corresponding child edges. Figure 4.2 illustrates the merge operation for the two main cases: (a) There exists an edge under node v_2 with a new symbol, in which case it will simply be copied over to the current merged tree node v_1. (b) There exist edges e_1 and e_2 under nodes v_1 and v_2 that share a common prefix, given as $e_1 = \alpha\beta$ and $e_2 = \alpha\gamma$, where $\alpha, \beta, \gamma \in \Sigma^*$, in which case we create a new internal node with edge label α, which in turn has two children with edge labels β and γ. For example if $e_1 = CAAT$ and $e_2 = CAGGC$, we get an internal node with edge label CA and with the two children shown in Figure 4.2(b).

Merging the tree incurs random accesses to the input string. We would like to emphasize that our target input sequences are genome-scale DNA sequences. Since DNA alphabet is $\{A, C, G, T\}$, we can encode/compress the input string using 2 bits per character. Since the largest genome that is publicly available is the human genome, we can expect to use up to $(3\times10^9)/4$ or 750MB of memory to hold the input string. Since this amount of memory is not unreasonable to assume on a modern computer, for our target input of large-scale DNA sequences, we assume that there is no need for a buffer management strategy for the input string. Note that even for genome-scale alignments between two whole genomes, only one string is needed to be kept in memory at any time. The query genome can be streamed through the disk-based tree for the first genome, and matching anchors can be output.

Note that our tree merging strategy differs from that of ST-Merge [35]. In ST-Merge, the subtrees are merged together (at once) to form a single complete suffix tree, whereas Trellis only merges the subtrees of the same variable-length prefix together, creating m persistent prefixed suffix trees in total. Each final in-memory prefixed suffix tree is written to the disk in a depth-first manner. For an internal node, we write the following information: the index into S denoting the start of the node's edge label, the edge length, and the disk locations of all five of the node's children. For a leaf node, we store the starting index of its edge label, and the suffix id that the leaf represents.

4.4 Suffix Link Recovery Phase

The existence of suffix links in a suffix tree is crucial to ensure efficiency in many fast string-processing algorithms, such as genome alignment via matching anchors [4, 12, 26] and tandem repeats search [23]. TRELLIS has an additional post-construction step that rebuilds the suffix links, if needed. This option enables existing string-processing algorithms to directly use our disk-based suffix tree.

Although Ukkonen's algorithm is used to create the prefixed suffix subtrees (T_{R_i,P_j}), not all internal nodes in the final merged prefixed suffix tree (T_{P_j}) have a suffix link. The merging phase may discard internal nodes (from the subtrees) with a suffix link and it may create new internal nodes (in the merged tree) without a suffix link. We found that only a small fraction of the nodes in the final merged tree (per prefix) have suffix links. We also found that completely disregarding them and recovering the links from scratch takes about the same time as keeping the original link information. Therefore, TRELLIS essentially ignores all of the suffix links after the partitioning phase, which reduces the amount of data the algorithm needs to keep track of per node. As a result, the amount of data read from and written to the disk is also reduced.

The pseudocode for suffix link recovery is given in Figure 4.3. The general idea of suffix link recovery is as follows. Recall that our suffix tree is in the form of many prefixed suffix trees T_{P_j} (for $j \in [0, m-1]$). TRELLIS recover the links for one prefix-subtree T_{P_j} at a time. It proceeds in a depth-first manner; starting from the children of the root, for any internal node, v, the algorithm locates v's parent, $p(v)$, as well as the parent's suffix link, $sl(p(v))$ (Figure 4.3, lines 2 – 10). This suffix link may point to another prefix-subtree T_{P_a}, which is loaded into memory the first time it is accessed (lines 7 – 9). Next the algorithm skip/counts [36] down from $sl(p(v))$ to locate $sl(v)$ (lines 11 – 15). Once found, the disk location (physical pointer) of $sl(v)$ is added to v (line 16), and the algorithm then proceeds recursively to all children of v that are internal nodes (lines 18 – 22). Note that all subsequent suffix links for children of v (in T_{p_j}) are guaranteed to be found under $sl(v)$ in T_{p_a}. Thus in a depth-first manner all the suffix links of T_{P_j} are recovered, bringing in additional trees T_{P_a} as required. Due to the depth-first traversal, the prefixes P_a are also accessed in

```
1  recoverSuffixLink (v);
   Input      : An internal node v
   Output     : None
2  if v is root then
3  |  sl(v) ← v;
4  else
5  |  p(v) ← parent of v;
6  |  l ← edge length between p(v) and v;
7  |  if Prefixed suffix tree of sl(p(v)) is not in memory then
8  |  |  Read the tree into memory;
9  |  end
10 |  sl(p(v)) ← node pointed to by suffix link of p(v);
11 |  if p(v) is root then
12 |  |  Skip and count down from sl(p(v)) for l-1 characters (minus the
   |  |  first character);
13 |  else
14 |  |  Skip and count down from sl(p(v)) for l characters;
15 |  end
16 |  sl(v) ← the internal node reached;
17 end
18 foreach v's child node u do
19 |  if u is an internal node then
20 |  |  recoverSuffixLink (u);
21 |  end
22 end
```

Figure 4.3: Suffix Links Recovery Algorithm

lexicographical order (since the children of each node are accessed in the array in a fixed order, e.g., A, C, G, T). For example, let the set of variable-length prefixes be $P = \{AC, CA, CC, CG, CTA, CTC, CTG, CTT\}$. The suffix links originating from T_{AC} may lead to T_{CA}, T_{CC}, ..., T_{CTT}. When recovering the links that originate from T_{AC}, we traverse T_{AC} in depth-first order. Thus, the suffix links from T_{AC} to T_{CA} will be rebuilt first, then to T_{CC}, T_{CG}, and so on. Finally, note that at any given time, the *internal nodes of at most two prefixed suffix trees* need to be accessed. Also note that the leaf nodes need not be accessed during this step because suffix links are not defined on them.

For efficient disk-based link recovery, TRELLIS avoids as many disk operations as possible. We found that loading the entire set of internal nodes for the needed two prefixed suffix trees into the main memory prior to performing the suffix link recovery results in several times the speedup over operating directly on the disk. For our example above, during the suffix link recovery of T_{AC}, T_{CA} will be loaded and discarded before T_{CC}. T_{CC} will be loaded and discarded before T_{CG}, and so on. T_{AC} is always kept in memory because it is the tree whose suffix links are being recovered. After all suffix links of T_{AC} are recovered, T_{AC} is written back to the disk (each internal node is augmented with its suffix link location). Since t is chosen such that the memory can completely hold the input string and two sets of internal nodes (see Section 4.5), this step can operate under the given amount of available memory.

4.5 Choosing the Right Threshold

Now that we have explained all the steps in TRELLIS we can describe how the value of t is chosen. Recall that t is the threshold used for creating the variable length prefixes as well as for the partition size. Let M be the available main memory. For a typical DNA sequence, the number of internal nodes in the suffix tree is about 0.7 times the number of leaf nodes, whereas there are exactly n leaves (one for each suffix).

The threshold value t in TRELLIS represents the bound on the total number of suffixes in the subtree(s) being operated in memory at any given time. Therefore, we must ensure that the memory needed to encode the string and to maintain the subtree(s) being worked on do not exceed the available memory M. Our implementation requires at most 40 bytes per internal node and 16 bytes per leaf node. Children of an internal node are stored in a fixed-size array of length $|\Sigma| + 1 = 5$. In addition, more internal nodes, on the order of $0.6t$, are created during the merging phase. Also, during the suffix recovery phase we need to keep access to internal nodes from two different prefixes. Thus during the construction of the disk-based tree we need to keep memory equivalent to approximately two sets of internal nodes (i.e., $0.7t + 0.7t = 1.4t$). Therefore, given the maximum amount of available memory

M (in bytes), t is chosen to be the largest value that satisfies the following:

$$M \geq \frac{n}{4} + ((1.4 \times 40) + 16)t \rightarrow t \leq \frac{M - n/4}{72}$$

Note that the $\frac{n}{4}$ term represents the space requirement for the compressed input string. Since we use a bit-encoding of each character which takes 2 bits instead of 1 byte, the compression ratio is 1/4. Given the amount of memory to use, the above equation is used to compute the appropriate value for t.

4.6 Disk Reading and Writing of Suffix Trees

Disk access plays an important role in persistent suffix tree construction. Despite the fact that the space requirement of a suffix tree can be as compact as about 9 bytes per character indexed [20, 29], the tree size quickly becomes a major bottleneck for larger sequences. For example, the suffix tree for the human genome would require 27GB of disk space at the very least.

To guarantee the efficiency of TRELLIS, one of our main goals when designing the algorithm was to minimize disk I/O time. The threshold t guarantees that the size of a subtree being operated on at any given time is bounded by t and that the subtree fits entirely in the available memory. The only times that TRELLIS accesses the disk are:

- In the partitioning phase, when the subtrees T_{R_i,P_j} (for all $0 \leq i < r$ and $0 \leq j < m$) are written to the disk.
- In the merging phase, when the subtrees written in the previous phase are read back. Note that in the partitioning phase, the trees are written in the order of their partition numbers, but in the merging phase they are read back in the order of their prefixes.
- At the end of the merging phase, when the subtrees T_{P_j} (for $0 \leq j < m$) are written to the disk.
- In the optional suffix link recovery phase, when we read the internal nodes for each tree T_{P_j}, and also read the internal nodes for other the trees accessed.

Besides these above occasions, the algorithm operates strictly in memory. Notice that TRELLIS does not have any complicated node buffer management policy like in most existing algorithms: it simply reads and writes any needed subtrees *in their entirety* from and to the disk and then operates on them in memory. This is possible because the size of subtree(s) residing in memory at any given time is bounded by t, which is computed according to the amount of memory available. The tree nodes are written (and therefore read back) in a depth-first manner. TRELLIS's disk access behavior (with read/writes of a subtree at a time) contrasts with those of other algorithms where each disk access involves some small number of nodes, depending on their page size.

Let us take a closer look at how TRELLIS accesses the disk. In the partitioning phase, there are $r \times m$ subtrees to be written to the disk. In the merging phase, there are $r \times m$ subtrees to be read, and m merged trees to be written to the disk. In total the disk is accessed $m(2r + 1)$ times in these two phases. The total amount of data read from and written to the disk is about one and two times the suffix tree size, which is a large amount of data. However, this disk access pattern has a certain benefit. As a general rule of thumb, to ensure the best performance possible, disk access should always be performed sequentially, using as few as possible file-system calls. TRELLIS adheres to this guideline and minimizes its disk I/O time by reading and writing a large block of sequential data, i.e. a subtree, at a time. Although this disk I/O scheme may not be the most efficient (in the merging phase, some of the nodes may be unnecessarily read back and then re-written to the disk after the merge.), performing disk operations in this manner is simple and sufficiently fast, such that it allows our method to outperform existing algorithms.

4.7 Computational Complexity

Before examining the empirical performance of TRELLIS, we consider its computational complexity. We discuss each phase separately below.

Prefix Creation Phase: Recall that the variable length prefix creation phase proceeds in steps. In each scan over the input string, only the frequencies for prefixes up to a certain length are computed. After each stage, only those prefixes

whose frequencies exceed the threshold t are extended. At the end we obtain a set $P = \{P_1, P_2, \cdots, P_{m-1}\}$ of m variable length prefixes, with each prefix having frequency at most t.

Let $l = \max_i\{|P_i|\}$ be the longest prefix length obtained. If l were known, and only a single scan were made over the input string S to compute the set P, then at each position in S we would have to update the frequencies of l prefixes (the substrings of lengths 1 through l starting at the given position), which would yield a time complexity of $O(nl)$, and a space complexity of $O(n + |\Sigma^{l+1}|)$. In practice, the multi-stage approach capitalizes on the pruning effect of the frequency threshold t, to reduce both the time and space requirements, at the cost of multiple scans of the input string S. For example, for the human genome (with $t = 10^6$) only two scans of S suffice (the entire input string is kept in memory).

Partitioning Phase: In the partitioning phase, the input string is broken into $r = \lceil \frac{n+1}{t} \rceil$ partitions (R_i, $i \in [0, r-1]$) and Ukkonen's algorithm is used to build a suffix tree T_{R_i} from each partition R_i. Since each partition is a subsequence of length (at most) t, each suffix subtree T_{R_i} takes $O(t)$ time/space to construct [36]. Finally, a single traversal of each T_{R_i} is required to split it into the prefixed subtrees T_{R_i,P_j} (for $j \in [0, m-1]$). Across all the r partitions the time/space complexity adds up to $r \times O(t) = O(n)$. Only one scan of S (which is kept in memory) is required in this step. Note that, as mentioned in Section 4.2, TRELLIS actually scans ahead $c \ll t$ characters to make the suffix tree of each partition explicit. For a pathological case (e.g., for a long string over a single character), this look-ahead approach can pose a problem; it is easy to avoid this pathological case by adding a "virtual" terminal symbol at the end of each partition to make all suffixes explicit. However, since such cases do not arise for real DNA sequences, we prefer the faster look-ahead approach.

In terms of disk I/O, in the partitioning phase, there are $r \times m$ subtrees T_{R_i,P_j}, each of size $O(\frac{t}{m})$, which are written to the disk. Since $r \times m \times \frac{t}{m} = O(n)$, the entire suffix tree is written to the disk once. This is done in $O(rm)$ disk accesses, since TRELLIS reads and writes an entire subtree in one step. Notice that TRELLIS does not have any complicated node buffer management policy like in most existing

algorithms: it simply reads and writes any needed subtrees *in their entirety* from and to the disk and then operates on them in memory. This is possible because the size of subtree(s) residing in memory at any given time is bounded by t, which is computed according to the amount of memory available. The tree nodes are written (and therefore read back) in a depth-first manner. The disk access behavior of TRELLIS (with read/writes of a subtree at a time) contrasts with those of other algorithms where each disk access involves some small number of nodes, depending on their page size.

Merging Phase: In the merging step, the subtrees from all partitions T_{R_i,P_j} ($i \in [0, r-1], j \in [0, m-1]$) are merged into the merged suffix tree T_{P_j} for each variable length prefix P_j. For each merge operation, we may have to traverse $|\Sigma| = 4$ edges at each node (which is negligible), and have to compare as many characters as the longest common prefix (LCP) of the edges being merged. Let p denote the average LCP across all merge operations. Thus the time for each merge is $4p = O(p)$. Across all the variable length prefixes the total merging time is $O(pn)$ since the number of tree nodes and edges in the full suffix tree is bounded by $O(n)$. Since $p = O(n)$ in the worst case, we have a total time complexity of $O(n^2)$. However, note that $p = O(n)$ is a very pessimistic bound on the average LCP. For example, we found that the average longest common prefix is about 30 for large DNA sequences, suggesting that on average $p = \log(n)$. Therefore, the complexity of the merge step is $O(n \log n)$ in practice, and is $O(n^2)$ in the worst case.

The space complexity of the merging step remains $O(n)$ across all prefixes, since in total there are $O(n)$ nodes in the full suffix tree and the space required for the input string is also $O(n)$. Note that several scans of the input string are required, since for each merge on average p characters from S are read.

The disk I/O complexity of the merging step comprises of $r \times m$ subtrees T_{R_i,P_j} (each of size $O(\frac{t}{m})$) that are read, and the m merged trees T_{P_j} (each of size $O(t)$) that are written to the disk. This is equivalent to reading and writing the entire suffix tree once. Note that the trees are read in $O(rm)$ steps, and written in $O(m)$ steps; each step reads/writes an entire tree.

Suffix Link Recovery Phase: In this phase we recover the suffix links for all

internal nodes in the final disk-based suffix tree, which has $O(n)$ nodes. At each internal node, we walk up one edge to find its parent node with a suffix link. We then follow the suffix link, skip/count down some number of nodes, and add the suffix link for the given internal node. It is easy to see that the first and last operations take constant time. Also it has been shown in [22, theorem 6.1.1], that over the entire suffix tree the skip/count steps takes at most $3n = O(n)$ time. (It can be adopted from the proof that, over the entire suffix link recovery step, "the current node depth is decremented by at most $2[n]$ times, and since no node can have depth greater than $[n]$, the total possible increment to the current node-depth is bounded by $3[n]$ over then entire [step]. Therefore, the total number of edge traversals during the down-walks is bounded also by 3[n]." Since the skip/count complexity over the entire step is $O(n)$, the complexity of this step is also $O(n)$.) Thus the total time complexity of suffix link recovery is $O(n)$. The space requirement remains $O(n)$ as well, since TRELLIS needs access to the input string, and internal nodes from at most two prefixed suffix trees at any given time.

In terms of the disk I/O cost, recovering the suffix links for a given prefixed suffix tree T_{P_j} requires that we keep all its internal nodes in memory. In addition, in its depth first traversal, we need to read the internal nodes from one other tree, say T_{P_a}, at any given time. As each tree T_{P_a} is needed it is read into memory (in one step), replacing any previous tree that may have been read. Note that even though all internal nodes are brought into memory, only the relevant internal nodes pointed to by some suffix link in T_{P_j} are actually used. Also, once the links are recovered the entire tree is written back to disk. The overall I/O cost is hard to characterize, but in the worst case, each prefixed tree may be read m times, once for each prefix P_j. This would mean $O(m)$ reads of the disk based tree in the worst case. In practice, however, each prefix tree T_{P_j} has links to only a few other trees T_{P_a}, requiring only a few tree loads per prefix. The writing of the merged trees after recovery of suffix links is equivalent to one complete write of the full tree.

Putting together the time and space complexity from the various phases, we find that the merging phase is potentially the most expensive due to its $O(n^2)$ worst case time complexity, which also gives TRELLIS its $O(n^2)$ worst case time complexity.

On average the time complexity of the merging phase, and hence Trellis, is closer to $O(n \log n)$. The space complexity of Trellis remains $O(n)$.

CHAPTER 5
OPTIMIZED TRELLIS:
FASTER AND MORE SCALABLE

In this chapter, we propose a faster and more scalable version of TRELLIS. For simplicity, we will refer to this version as TRELLIS-2 throughout the rest of the thesis. Our contributions in this chapter are two-fold. First, we speed up the original TRELLIS by making possible larger suffix subtrees in the partitioning phase while using the same amount of memory. Second, we introduce a combination of techniques that together makes a novel string buffering strategy for the merging phase. The buffer allows TRELLIS-2 to assign a very small amount of memory for the input string, and as a result, removes the limitation of the original TRELLIS. Our method can now construct external suffix trees for input sequences whose size is much larger than the available memory within a reasonable amount of time. Details of the optimizations are as follows.

5.1 Original TRELLIS

Let us first recap the TRELLIS algorithm. The algorithm overview is shown in Figure 4.1 of the last chapter. Given an input sequence S, we first segment it into several partitions R_i (step a). The partition size is chosen such that the resulting suffix tree T_{R_i} from each partition (step b) fits in the main memory. Each resulting suffix tree is further split into smaller subtrees T_{R_i,P_j}, that share a common prefix P_j, which are then stored on the disk (step c). After the partitioning phase, the subtrees T_{R_i,P_j} for each prefix P_j are merged together into a merged prefix suffix subtree T_{P_j} (step d). In contrast to other algorithms, TRELLIS uses variable-length prefixes which ensures roughly equal prefix suffix subtree sizes (sizes of T_{P_j}). The prefixes are chosen such that their suffix subtrees also fit entirely in memory. After each subtree T_{P_j} is constructed, it is written to disk. The complete suffix tree is simply a forest of these prefix-based subtrees (T_{P_j}).

The cornerstone of the TRELLIS algorithm is the condition that the size of the subtree being operated in memory at any given time plus the size of memory required for the string must never exceed the size of the available memory. This includes the size of T_{R_i} and $|R_i|$ during the partitioning phase and the size of T_{P_j} and the *entire* input string during the merging phase. Note that additional internal nodes are created due to the subtree merging, so one must take into account extra space required for these nodes as well. To ensure that the above memory condition is met, TRELLIS relies on the threshold t which bounds the maximum number of nodes that can be present in the current subtree being worked on in memory.

The value of t is chosen as follows. Let M bytes be the available main memory. Let $Size_i$ and $Size_l$ be the size of an internal and leaf node. For a typical DNA sequence, the number of internal nodes in the suffix tree is about 0.6 – 0.8 times the number of leaf nodes. Using t as the partition size in the first step, there can be at most t leaves (one for each suffix) in each T_{R_i}. Therefore, during the partitioning phase, t must satisfy the following equations:

$$M \geq \frac{t}{4} + ((0.8 \times Size_i) + Size_l)t \tag{5.1}$$

$$t \leq \frac{M}{\frac{1}{4} + ((0.8 \times Size_i) + Size_l)} \tag{5.2}$$

The $\frac{t}{4}$ term represents the space requirement for the compressed input string. Our target input sequences are genome-scale DNA sequences whose alphabet is {A, C, G, T}, the string can be encoded/compressed using 2 bits per character. Since each character takes 2 bits instead of 1 byte, the compression ratio is 1/4.

During the merging phase, the threshold t ensures that T_{P_j} fits under M bytes. Variable-length prefixes are chosen such that the frequency of each prefix does not exceed t. This means for any given T_{P_j}, there are at most t leaves and $0.8t$ internal nodes. As noted earlier, additional internal nodes on the order of $0.6t$ are created from tree merging. Therefore, in the merging phase, t must satisfy the below equations:

$$M \geq \frac{n}{4} + ((1.4 \times Size_i) + Size_l)t \tag{5.3}$$

$$t \leq \frac{M - \frac{n}{4}}{((1.4 \times Size_i) + Size_l)} \tag{5.4}$$

Note that in this equation, we use the term $\frac{n}{4}$ instead of $\frac{t}{4}$ because merging incurs random accesses to the entire input sequence.

Since the t value resulting from equation (5.4) is always smaller than the value from the equation (5.2), we use equation (5.4) to determine t in TRELLIS. This guarantees that any subtree being worked on at any given time (either during partitioning or merging phase) can reside entirely in memory.

5.2 Optimized TRELLIS: Faster and More Scalable

In this section, we introduce two optimizations to the original TRELLIS. The first optimization is based on a simple observation that *larger* suffix subtrees can be created in the partitioning phase under the *same* memory restriction. As a result, there is less disk management overhead, and fewer merge operations are required, speeding up the algorithm. The second optimization is a novel string buffering strategy. The buffer is based on several techniques, which together remove the limitation of TRELLIS that requires the input sequence to fit entirely in memory. This means our algorithm can index sequences that are much larger than it could before, i.e. $n \gg 4M$. With these optimizations implemented, TRELLIS-2 becomes even faster and much more scalable than its original version.

5.2.1 Optimization 1: Larger Partition Size

To calculate the value of t in TRELLIS, we use the smaller of the two values resulting from equations (5.2) and (5.4) above. Equation (5.4) always yield a smaller t because more internal nodes and the entire input sequence need to be kept in memory. However, recall that the goal here is merely to ensure that the tree being operated in memory at any given time fits under M. Our first optimization is based on a simple but effective observation that the partitioning phase need not use the smaller t value. Equation (5.2) already guarantees that each suffix subtree T_{R_i} always fits under M. To guarantee that each T_{P_j} in the merging phase fits under M, the variable-length prefixes are still created based on the t value from

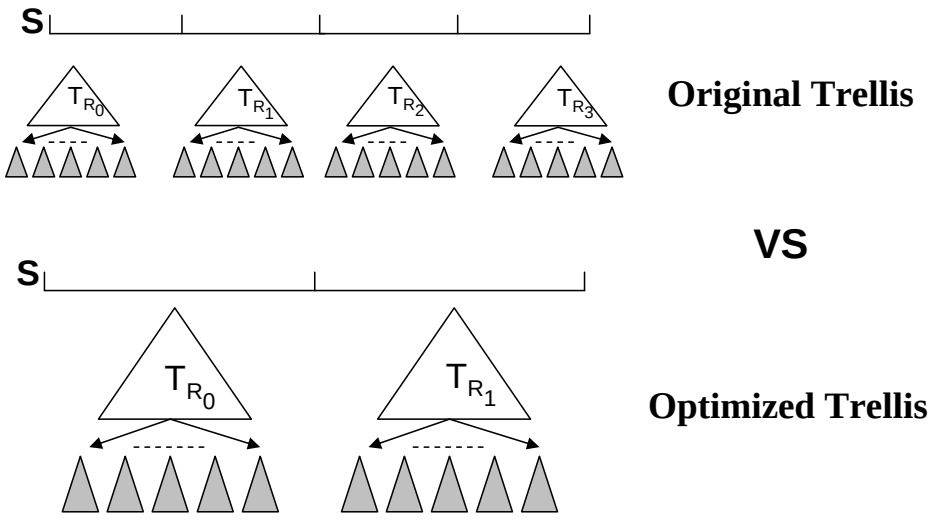


Figure 5.1: Optimization 1: Larger partition size speeds up the algorithm by lowering the disk overhead and number of merge operations required.

equation (5.4). The illustration of this optimization is shown in Figure 5.1. The difference between TRELLIS and TRELLIS-2 is that the partition size is larger and there are fewer partitioned suffix subtrees in TRELLIS-2. Note that the same variable -length prefixes are used in both algorithms because the prefixes are created based on the merging phase threshold.

With the larger partition size, larger subtrees T_{R_i} can be created. Each subtree is split (step c, Figure 4.1) based on the same set of variable-length prefixes as in its previous version. The main improvement over TRELLIS is that less merging operations (step d, Figure 4.1) are required per prefix due to fewer and larger partitioned suffix subtrees to begin with. More importantly, a larger partition size incurs less disk-write overhead (step c, Figure 4.1), i.e. fewer overall prefix searches prior to writing the subtrees to disk because there are fewer subtrees, fewer disk seek operations, and less internal hard disk fragmentation for the operating system to resolve because there are fewer subtree files to manage. Experimental results in the next chapter show that as the input sequence length increases, the t value resulting from equation (5.2) becomes much greater than the one resulting from equation (5.4) which allows this simple optimization to consistently speed up the

algorithm. The speedup becomes especially more pronounced when the input sequence size increases, i.e., as n becomes very close to $4M$.

5.2.2 Optimization 2: String Buffer

As previously stated, our goal is to be able to index genome-scale DNA sequences under a reasonably limited amount of memory. Our ultimate target sequence was the human genome because it is currently one of the largest completely sequenced genomes available. In [32], we show that TRELLIS can index it within about 4 hours using 2GB of memory. Nevertheless, there exist other larger genomes which should become available in the future, e.g. the 670 Gbp Amoeba dubia and 290 Gbp Amoeba proteus genomes [1]. Also with rapid sequencing technology, a vast amount of short sequence reads can be collected, which collectively represent very large sequence data. Such long sequences may present a problem for TRELLIS since it requires the compressed input sequence to fit completely in memory during the merging phase, to minimize incurring random accesses to the input string. To prepare for extremely large sequences, we optimize our algorithm by using a novel string buffering technique, which requires a much smaller amount of memory to be assigned to the input string buffer and yet is able to index large sequences in a reasonable amount of time.

The string buffering strategy relies on several different techniques, each uniquely important because of its impact on the buffer hit rate. For the sake of simplicity, the combined scheme in this section will be referred to collectively as the buffer strategy for the rest of the thesis. The basic idea behind the buffer design is to keep the characters most likely to be accessed in memory and load the rest from disk on an as needed basis. Although suffix trees merging incurs random accesses to the input string, the following techniques adjust the string access patterns to improve locality and help retaining characters most likely to be accessed in memory.

Internal Edge Index Shifting The goal of our first technique is to condense the character accesses during suffix tree merging in a small region of the input sequence. This small region of the input string can then be kept in memory as a part of the string buffer, hence increasing the buffer hit rate. Details of this technique are as

follows.

A suffix tree edge is represented by two indexes $[start, end]$ denoting its edge label $S[start \ldots end]$. The basic idea behind this technique lies on an observation that these indexes need not be unique as long as they denote the same string label. For example, an edge with label "AT" may use the indexes $[0, 1]$ or $[1000, 1001]$ to encode its label as long as $S[0] = A, S[1] = T, S[1000] = A$, and $S[1001] = T$. Another important observation is that the edge lengths between two internal nodes, i.e. internal edge lengths, are generally short. In all of our experiments, most internal edge lengths fall between 1 and 25 characters. Within this group, the majority are only a few characters long as shown in Figure 5.2.

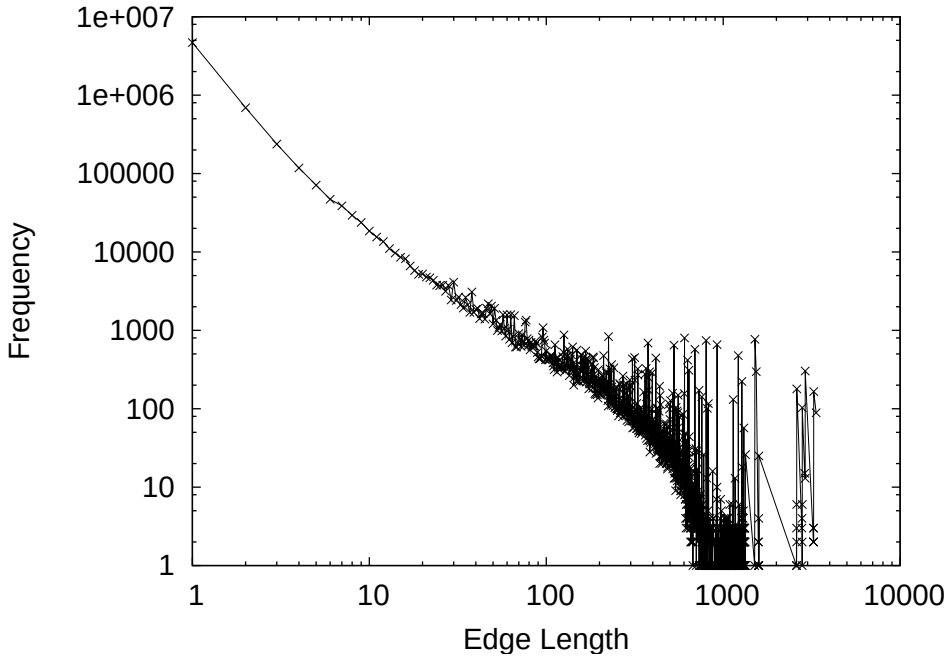


Figure 5.2: Frequency of internal edge length. The majority of internal edges have very short length (between 1 – 25 characters). The data were gathered from the first partition suffix tree of the first 200Mbp of human genome. The memory usage was 512MB.

To implement this technique, a small suffix tree of size 2 Mbp is independently maintained throughout the program. We use the first 2 Mbp of the human chromosome I as its input string. Prior to writing each internal edge in any T_{R_i} to the disk, we search for its string label in the small suffix tree. If found, we switch the edge's

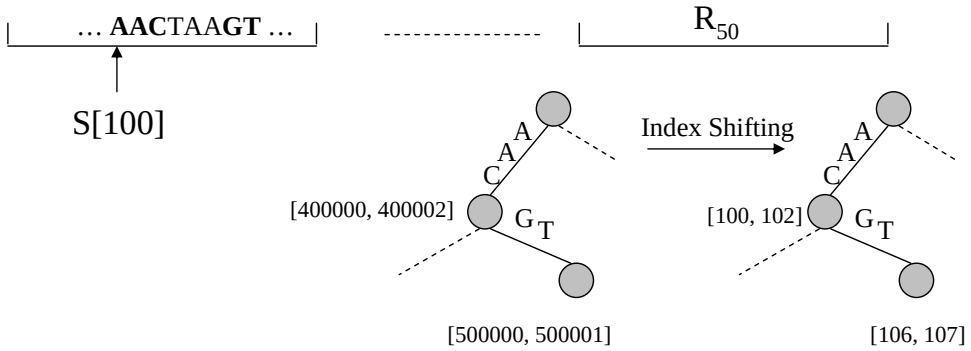


Figure 5.3: Internal Edge Index Shifting. In this figure, two edges from the 50^{th} partition have their indexes shifted to the lower range at the beginning of the input string.

We found that 97% of edge label indexes on average can be shifted to the range $[0 \ldots 2 \times 10^6)$ via this method. Figure 5.4 shows the data gathered from indexing the human genome using 512MB of memory. The graph indicates that the majority of the internal edge labels across all partitions were consistently found in the small suffix tree. Such behavior is due to the DNA small alphabet size and its distribution in nature which consists of numerous short repeats. Since the internal edge labels are very short, it is very likely that they can be found in most regions of the human genome. We randomly choose to use the first 2Mbp of the sequence and found that it works well. The string $S[0 \ldots 2 \times 10^6)$ is stored in the memory as part of the buffer because it will be heavily accessed during the merge. The memory required to keep the string in memory is 0.5MB (with string compression).

Keeping Leftover Ranges Approximately 3% of internal edge labels are not found in the small suffix tree, we store these leftover pairs of internal edge indexes in memory. During the merging phase, the strings corresponding to these index ranges are loaded into the main memory. These strings are also compressed using 2 bits per character. In all of our experiments, the memory required to keep these substrings ranges from a few MB to at most 20MB.

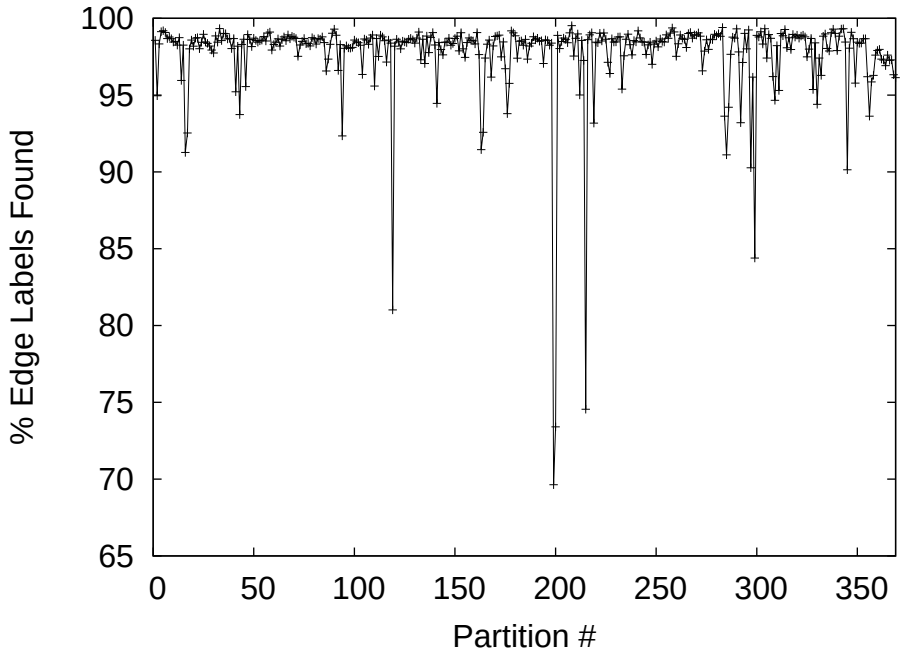


Figure 5.4: Approximately 97% of the internal edge labels of the human genome suffix subtrees across all partitions can be found in the small suffix tree. This behavior is due to the nature of DNA sequence; small alphabet size ($\Sigma = 4$) and a large number of short repeats.

Merged Edge Index Shifting As mentioned previously, additional internal nodes are created during the subtree merging phase. We also shift these index ranges to be in the $[0 \ldots 2 \times 10^6)$ range using the same approach as above.

Storing Last Partition String in Memory Subtrees are always merged starting from the 0^{th} partition to the last partition. When the i^{th} subtree is merged with the already merged subtree of the 0^{th} to $(i-1)^{th}$ partitions, the string of the i^{th} partition is more heavily accessed than the strings of the first i partitions. Therefore, it is wise to always keep the i^{th} partition string in memory when its subtree is being merged to the current merged subtree. The memory required for storing the last partition string is $\frac{t}{4}$ bytes.

Leaf Edge Label Encoding : It is impractical to also apply index shifting to leaf edges because there are more of them and they are generally magnitudes longer

than their internal counterparts. However, we discover that only a small amount of characters at the beginning of the leaf edges are accessed during merging. This is because leaves are relatively deeper in the tree and lengthy exact matches do not occur too frequently in nature. Therefore, merging does not require too many of leaf character accesses. To guarantee that the more frequently accessed characters are readily in memory, we allow 64 bits to store the first 29 characters (which require 58 bits, with 2 bits per character) of the leaf label. The last 6 bits are made into a counter to keep track of the characters that are no longer valid for this leaf edge, which may happen if an internal node is created as a result of merging this leaf edge with another edge. The encoded strings are stored with their respective leaf node, and not actually with the buffer. Since disk accesses are expensive, the encoded strings are loaded on as needed basis (we found that 15 – 35% of leaves are not accessed at all during the merge). The memory required for leaf edge label encoding is at most $8t$ bytes per prefix. We found that about 93 – 97% of leaf characters accessed during the merge can be found using the encoded labels.

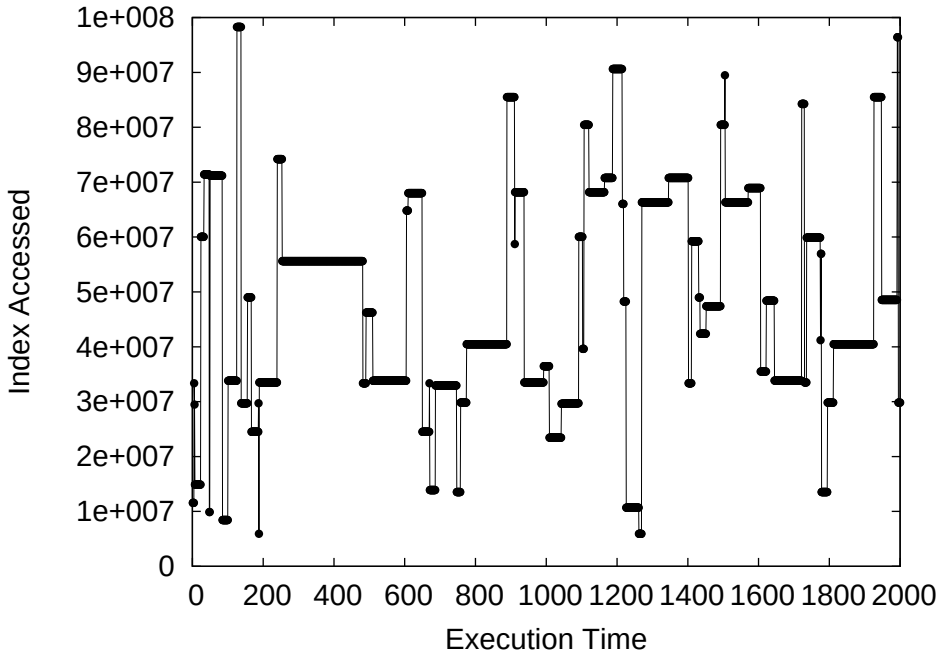


Figure 5.5: The string disk access pattern resulting from buffer misses exhibits poor locality of reference, with the exception that short subsequent ranges of characters are always accessed together.

As for the rest of the characters that are a buffer miss, they are directly read from the disk. We found that the input sequence's disk access pattern resulting from the buffer misses during the merge has very poor locality of reference, i.e. it is almost completely random, with the exception that short subsequent ranges of characters are accessed together (Figure 5.5). These short ranges represent the string of the edges being merged. Therefore, we keep a small buffer of size 256KB to store the characters that require a direct disk access: each disk read reads 256KB consecutive characters at a time. Lastly, when calculating the t values to use during the partitioning and merging phases, we must also deduct the memory required to keep track of the small suffix tree from the overall available memory M. In our implementation the small suffix tree takes about 70MB.

The total amount of memory required for the string buffer can be calculated by adding the amounts of memory required for each technique: 0.5MB for the *edge index shifting*, 20MB for the *left over ranges*, $\frac{0.25t}{(1024\times1024)}$MB for the *last partition*, $\frac{8t}{(1024\times1024)}$MB for *leaf edge label encoding*, and 0.25MB for the *small buffer*. In total, the buffer requires $(0.5+20+\frac{8.25t}{(1024\times1024)}+0.25)$ or $(20.75+\frac{8.25t}{(1024\times1024)})$MB. The amount is typically much smaller than $\frac{n}{(4\times1024\times1024)}$MB, which is the memory required for the input string in the original TRELLIS. The string buffer hit rates are reported for several combinations of buffering techniques in Figure 5.6. The pseudocode for the buffer is given in Figure 5.7. In summary, when a character is requested by the merging function, we first check whether it is in the last partition, in the first 2Mbp string, in the small 256KB buffer, encoded with the leaf edge or in the left over ranges, in that order. If all of these fail, then the character is fetched directly from the disk.

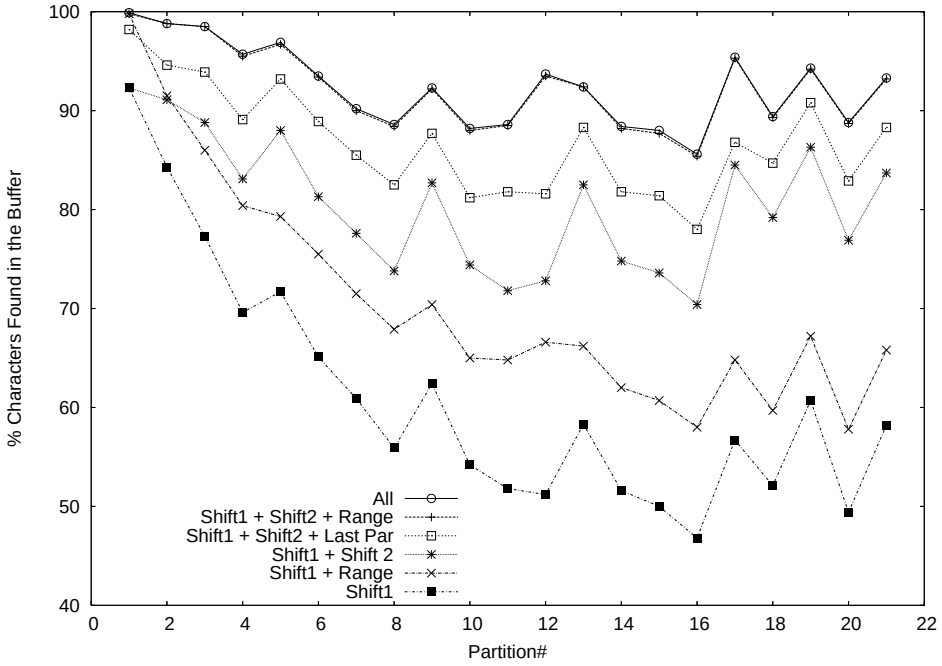


Figure 5.6: String buffer hit rates of various combinations of the buffering techniques. The data were gathered from indexing the first 200Mbp of the human genome using 512MB of memory. 21 partitions were required. *All* means all techniques were implemented. *Shift1* and *Shift2* mean the first and second internal index shiftings were implemented. *Last Par* means the i^{th} partition string was kept in memory. *Range* means the left over ranges were loaded in memory during the merging phase. The top two curves are almost identical. The difference between them is whether the last paritition is stored in memory. We found that storing the last partition in memory provides a slightly better buffer hut rate, but since a partition does not consume a lot of memory, we decided to keep it.

```
1  getCharAt (i);
   Input        : An input string index i
   Output       : S[i]
2  if i is in the last partition then
3  |  Fetch S[i] from the last partition string;
4  |  return;
5  end
6  if i < 2 × 10^6 then
7  |  Fetch S[i] from the S[0 ... 2 × 10^6) string;
8  |  return;
9  end
10 if i is in the small buffer then
11 |  Fetch S[i] from the small buffer;
12 |  return;
13 end
14 if i is available from a leaf edge then
15 |  Fetch S[i] from the encoded string of that leaf;
16 |  return;
17 end
18 if i is in the left-over ranges then
19 |  Fetch S[i] from the left-over range strings;
20 |  return;
21 end
22 Fetch S[i] from disk;
23 return;
```

Figure 5.7: String Buffer's getCharAt function

CHAPTER 6
EXPERIMENTAL RESULTS AND DISCUSSION

In this chapter, we study the performance of our method using two categories of experiments.

1. TRELLIS vs different state-of-the-art external suffix tree construction methods. Both construction times and query times are studied.

2. TRELLIS vs TRELLIS-2. The construction time between the two algorithms are compared in order to examine the effects of the optimizations.

6.1 TRELLIS vs TOP-Q, DynaCluster, and TDD

First, we compare our approach with different state-of-the-art external suffix tree construction methods. Since TRELLIS includes an optional step that fully reconstructs suffix links in the disk-based tree, we separated our experiments into two categories. First, we compared TRELLIS with the best known algorithms that maintain suffix links, namely TOP-Q [3] and DynaCluster [9]. Then, we compared it with the best known algorithms that do *not* maintain suffix links, namely TDD [34, 35] and ST-Merge [35]. Note that TDD and ST-Merge have been shown to outperform the former two algorithms by substantial margins. The results reported here are consistent with those reported in [32].

All of the experiments in this section were performed on a Linux machine with 2.2 Ghz Dual Core AMD Opteron processors, 1024KB cache, 32GB of RAM, and 4 high-speed Ultra SCSI 320 disks, each with 288GB disk space (for a 1.1TB total disk space). The maximum amount of RAM usage across our experiments was restricted to 2GB for the Human Genome case and 512MB for the rest of the cases. Note that the memory cap applies to all internal data structures including those for the suffix tree, memory buffers and the input string. TRELLIS was written in C++ and compiled with the GNU g++ compiler version 3.4.3 with optimization

flags activated. DynaCluster (executable only), TOP-Q, and TDD source codes were obtained from their respective authors and compiled under the same settings.

6.1.1 TRELLIS vs. TOP-Q and DynaCluster

As previously described, both TOP-Q and DynaCluster create trees having suffix links and do not exhibit the partition skew problem. Each achieves this differently. TOP-Q augments Ukkonen's in-memory suffix tree method using an effective buffer management strategy. As a result it retains all suffix links in the original Ukkonen's method, and it directly constructs the entire suffix tree (since there are no partitions, there is no partition skew problem). DynaCluster, on the other hand, clusters together the nodes that are likely to be co-referenced frequently, and constructs the suffix tree (without suffix links) in a depth-first manner. Due to the dynamic clustering strategy, DynaCluster does not suffer from the partition skew problem. It then uses a separate phase to rebuild the full set of suffix links in the tree.

Table 6.1: Suffix tree construction times for TRELLIS, DynaCluster and TOP-Q.

Input Length (Mbp)	TOP-Q (mins)	DynaCluster (mins)	TRELLIS (mins)
20	54	4	1
40	448	7	2
60	-	12	4
80	-	15	5
100	-	19	6

Since DynaCluster suffix tree construction phase is insensitive to the amount of memory available, in our experiments we employed their default buffer sizes for the tree construction. We used their recommended Lowest Common Ancestor (LCA) method of rebuilding the suffix links with buffer size of 512MB. The results reported for DynaCluster in Table 6.1 are the total times taken to construct the trees and rebuild the suffix links. It can be seen that TRELLIS outperforms DynaCluster by a factor of 3. On the other hand, when we ran DynaCluster on a 200 Mbp sequence (the first row in Table 6.2), it was running for over 8 hours, when we terminated

it. In contrast, on the same 200 Mbp sequence, TRELLIS takes only 13 minutes (to build the tree and to recover the links).

For TOP-Q, we used a buffer pool of 384MB for internal nodes and 128MB for leaf nodes. As suggested in their paper, the number of internal nodes for typical DNA sequences is 0.6 – 0.8 times the number of leaf nodes, and thus the memory was allocated accordingly to achieve the best performance possible. As shown in Table 6.1 the time taken for the smallest input string (20 Mbp) was already 54 minutes, whereas TRELLIS only took 1 minute. For 40 Mbp, TOP-Q took 448 minutes (7.5 hours), whereas TRELLIS took only 2 minutes.

Table 6.2: Suffix tree construction times for TRELLIS and TDD

Input Length (Mbp)	Memory Available (MB)	**TDD**	TRELLIS		
		Construction (mins)	Construction (mins)	Link Recovery (mins)	Total (mins)
200	512	9	11	2	13
400	512	64	24	4	28
600	512	92	41	7	48
800	512	213	64	9	73
1000	512	372	91	12	103
3000 (Human Genome)	2048	12.8hrs	4.2hrs	1.7hrs	5.9hrs

Table 6.3: Disk Space Usage: TDD vs. TRELLIS

Data Size (Mbp)	Trellis		TDD	
	Total (GB)	Bytes/Char	Total (GB)	Bytes/Char
200	5.0	26.8	1.8	9.5
400	10.1	27.0	3.6	9.7
600	15.1	27.0	5.4	9.7
800	20.2	27.0	7.2	9.7
1000	25.2	27.1	9.0	9.7
3000 (Human Genome)	71.6	27.1	54.0	19.3

6.1.2 TRELLIS vs. TDD

We next compared TRELLIS with TDD, the current state-of-the-art method for disk-based suffix tree construction. Experimental results are shown in Table 6.2.

Table 6.4: Memory-based Suffix Tree Construction

Input String (Mbp)	Running Time		Memory Usage	
	TRELLIS (mins)	TDD (mins)	TRELLIS (GB)	TDD (GB)
20	0.32	0.35	1.7	1.2
40	1.10	1.10	3.4	2.3
60	1.49	1.55	5	3.4
80	2.32	2.29	6.8	4.6
100	3.12	3.11	8.5	5.7

Prior to our work, TDD was the only algorithm that was reported to scale up to the human genome level. TDD does *not* maintain suffix links, which are crucial in many string-based problems as mentioned previously.

For the first 5 cases of the experiment, we restricted the amount of physical memory available to both algorithms to 512MB. The input sequences were the first 200, 400, 600, 800, and 1000 Mbp of the human genome, respectively. For the full human genome sequence, the memory was restricted to 2GB. For TRELLIS, we report separately the time taken to construct the suffix tree without suffix link recovery, for a fair comparison with TDD, which does not maintain suffix links. We also report the suffix link recovery time, and the total time for TRELLIS. Note that, as mentioned above, since DynaCluster was not competitive even on the 200 Mbp test case (it was running for over 8 hours, whereas TRELLIS finished in 13 mins), we did not do any further comparison with it.

Comparison with TDD: Experimental results in Table 6.2 show that TRELLIS outperforms TDD for all cases, except the 200Mbp sequence, where TDD is slightly faster. This is due to the fact that TDD uses four different buffers; string, suffix, temp and tree buffers. The 200Mbp string only required the temp and tree buffers, but the rest also required the suffix and/or string buffers. The use of these suffix/string buffers incurs additional disk I/O overheads and this is why TDD becomes slower for strings longer than 200Mbp. The total time taken for TRELLIS to 1) construct the tree and 2) both construct the tree *and* recover the suffix links were, respectively, between 2.2 - 4 and 1.9 - 3.6 times faster than the time taken by TDD just to construct the tree. For the entire human genome ($\approx$ 3Gbp) TDD took

12.7 hours to construct the tree while TRELLIS took 4.2 hours to construct the tree and an additional 1.7 hours to rebuild the suffix links. The total time for TRELLIS to construct the tree and rebuild the suffix links for the entire human genome is more than *two times* faster than the total time TDD took to construct the tree.

Compressed String/Suffix Tree: Note that TRELLIS applies the bit-encoding method to compress the input sequence, and thus the amount of memory required to encode the entire human genome for TRELLIS is about $3GB/4 = 0.75GB$. This means TRELLIS does not require any disk access for the input string during the tree construction, whereas TDD does (provided the available memory is less than 3GB). At first glance, it may seem that the advantage of TRELLIS over TDD may be due to the string compression scheme. However, that is not the case. While TRELLIS compresses the input string and obtains a 1/4 reduction, TDD uses a more efficient (compressed) suffix tree representation that uses about 1/3 – 1/4 less space (see bytes/char in Table 6.3) in memory. For the sake of completeness, we also experimented with giving both algorithms proportional buffer space besides the input string, and the results obtained support our claim. For example, for the 600Mbp string, restricting the memory limit to 1GB, we gave TDD 600MB (for input string) plus 400MB (for other buffers) for a total of 1GB memory. We assigned TRELLIS 600MB/4 = 150MB (for compressed string) plus 400MB (for all other in-memory structures) for a total of 550MB memory (roughly half of that given to TDD). TRELLIS took 43 mins whereas TDD took 58 mins to construct the suffix tree for the 600Mbp string. This confirms that the TRELLIS strategy is effective for disk-based cases.

Disk Space Usage: Table 6.3 shows the disk usage in terms of the size of the disk-based suffix tree for both TRELLIS and TDD. We report the total size (in GB) for the full suffix tree, as well as the number of bytes per character indexed. For the smaller sequences, TRELLIS consumes 2.8 times more disk space than TDD. This is because TDD is built using the memory optimized wotd-eager suffix tree method [20]. Nevertheless, note that for the full human genome, TRELLIS consumes

only 1.3 times the disk space used by TDD. This is because we had to run TDD in 64-bit mode for the full human genome, and it thus took twice as much memory as in the 32-bit mode. On the other hand TRELLIS was run in 32-bit mode throughout. We conclude that for smaller strings the memory optimized TDD method has an advantage. However for genome-scale strings both TRELLIS and TDD are roughly comparable (though TDD still has an advantage) in terms of disk space.

Memory-based Results: For sanity check, we ran both TRELLIS and TDD without setting any limit on the memory usage. Thus the entire 32GB of RAM was made available to both methods. Note that for this memory-based case, TDD becomes equivalent to the wotd-eager [20] method, and TRELLIS becomes equivalent to the Ukkonen's [36] method, since there is only one partition consisting of the entire input string, which yields the full suffix tree (and the merging/suffix recovery phases are not required).

Table 6.4 shows the running time and memory usage for TRELLIS and TDD on small input strings. Here the entire suffix tree fits easily in memory. Comparing with Table 6.1 we find that the memory-based TRELLIS is about 2 times faster than the disk-based TRELLIS on smaller strings. We also find that both TRELLIS and TDD have comparable performance in terms of time. However, note that for 100 Mbp input, TRELLIS is already consuming 8.5GB of memory, whereas TDD is consuming about 6GB. When we ran them on a 200 Mbp string, both methods finished in about 9 minutes and consumed about 18GB of memory[3]. When we ran the memory-based methods on an input string of 400 Mbp, we found that both methods started to thrash; they were using up in excess of 30GB of memory, and the CPU utilization went down from 99.9% to around 5%. We terminated the runs due to the excessive amount of thrashing. This underscores the importance of effective disk-based genome scale suffix tree methods, since even with a relatively powerful machine with 32GB of RAM, we were not able to construct a suffix tree for even a 400 Mbp input string. Furthermore, the full suffix tree for the entire human genome requires size in excess of 71GB and 54GB on disk, for TRELLIS and TDD,

[3]TDD was run in 64-bit mode for >200 Mbp strings, since its internal data structures could utilize the 32GB RAM only in 64-bit mode.

respectively (see Table 6.3). The in-memory construction consumes even more memory due to additional book-keeping overheads. Thus it is clear that memory-based methods are not able to successfully build the entire suffix tree at the genome scale.

Comparison with ST-Merge: The ST-Merge algorithm [35] is a variant of the TDD algorithm. The main bottleneck of TDD is the random access to the input string, which translates to heavy disk I/O overhead when the input string is very large. ST-Merge improves upon this aspect of TDD by using a partitioning scheme that causes the program to have more spatial locality of reference for the input string. Note that we were not able to compare timing results directly with ST-Merge since a working executable/code is not currently available from its authors[4]. Nevertheless, as reported in [35], ST-Merge starts to outperform TDD *only* when the input string is *significantly* larger than the amount of available main memory. When the input string fits entirely in the main memory, ST-Merge is exactly the same as TDD. It was observed in their experiment that when the input string was *at least three times* larger than the memory, ST-Merge started to outperform TDD. Since the human genome is about 3GB in length and the available memory in this case was 2GB, the human genome is not large enough for ST-Merge to benefit from its partitioning scheme and outperform TDD. Also note that as shown in Table 6.2, Trellis outperforms TDD by a factor of 2.2 on the entire human genome including the time it takes to recover the suffix links. ST-Merge, like TDD, does not maintain suffix links, whereas Trellis is able to recover the complete set of suffix links in under 2 hours, making it suitable for many string-based algorithms [22]. Due to the above reasons, Trellis has a clear advantage over ST-Merge and is the algorithm of choice for constructing suffix trees of large genome-scale DNA sequences.

[4]Personal communication.

6.1.3 Query Times: TRELLIS vs. TDD

We also compared TDD and TRELLIS in terms of query times using the disk-based tree. We built a disk-base suffix tree for the entire human genome. The memory was restricted to 2GB for both programs. 500 random queries of different lengths ranging from 40bp to 10,000bp were used to evaluate the methods. The queries were searched directly on disk. The minimum length of 40 was chosen because it is a typical minimum match/anchor size used in genome alignment programs, such as in MUMmer [12]. The queries were substrings of the human genome and randomly chosen across the whole sequence. Both programs were fed the same set of queries. Also, for a fair comparison, suffix links were not used for TRELLIS. All queries search for a match starting at the root of the disk-based suffix tree in both methods.

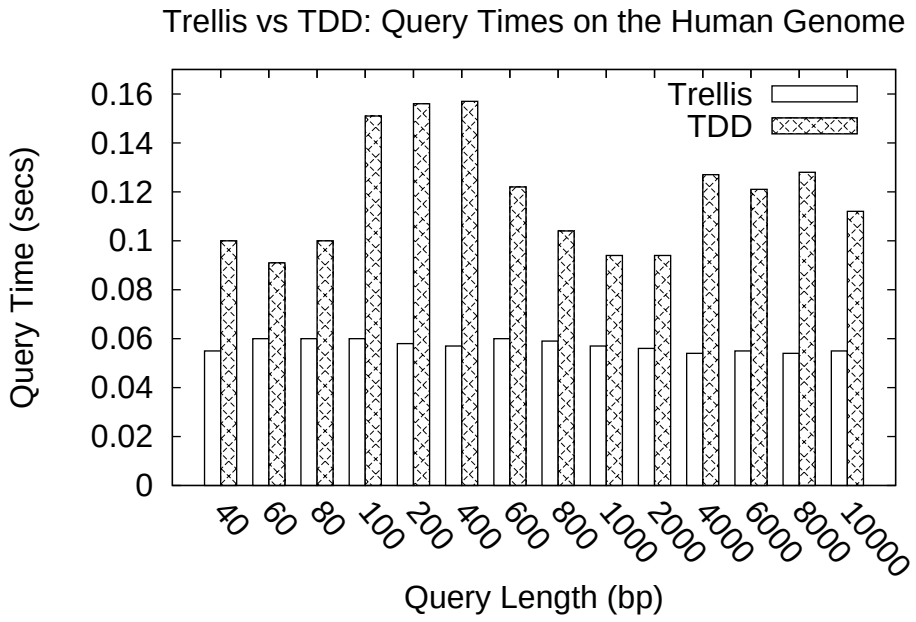


Figure 6.1: Average times of 500 random queries on the human genome suffix tree.

Figure 6.1 shows the average query times of the 500 random queries. We observe that on average TRELLIS is about 2 - 3 times faster than TDD. It is worth remarking that the average query time for even the longest query (with length 10,000bp) took under 0.06s for TRELLIS, showing the benefits of suffix tree indexing.

These query times also represent a disk-space versus query-time tradeoff. For TDD, the on-disk suffix tree size for the human genome is about 19 bytes per character indexed, whereas for TRELLIS the disk overhead is about 27 bytes per character (see Table 6.3). In the disk-based suffix tree constructed by TDD (as well as for ST-Merge), the length of an edge label is not stored with the edge itself, but instead can be determined by examining the children of the current node. In addition, each internal node only has a pointer to its first child, i.e., the children of an internal nodes must be linearly scanned during a query search. In contrast, TRELLIS stores the edge length with its respective node and each internal node has all of its children's location stored locally. Therefore, during a query search, TRELLIS requires fewer disk seeks, which result in a faster query time. Clearly, faster query times are desirable, even at the expense of some additional disk space, since disk-space is relatively cheap and abundant.

6.1.4 Query Times: With and Without Suffix Links

In this section, we investigate the effect of suffix links on the query times. Accesses of suffix links in this experiment were designed to imitate those incurred by an exact match alignment anchor finding algorithm used in a typical alignment program (see [12] for details). To briefly describe the algorithm, let $Q = q_0 q_1 \dots q_{|Q|-1}$ be a long query sequence for which we would like to find all exact match anchors against S. This can be done by streaming Q against the suffix tree of S. Starting from q_0, the algorithm matches Q with the tree for as far as possible. Let v be the last internal node encountered during the match, with label $\sigma(v) = q_0 \dots q_i$, and let q_j be the last matched character (i.e., the substring $q_0 \dots q_j$ has been found in S). To find the next match, i.e., the match starting at q_1, we can use the suffix link $sl(v)$ of v to avoid matching characters from $q_1 \dots q_i$. Instead we can skip/count down from $sl(v)$ to match $q_{i+1} \dots q_j$, and then we can search for the remaining characters starting at q_{j+1}. Subsequent matches can be found in a similar manner.

We ran query experiments on the entire human genome suffix tree (see Figure 6.2). Several query lengths, ranging from 40bp to 10,000bp, were tested. For each length, 500 consecutive queries were issued. Given a random starting index i

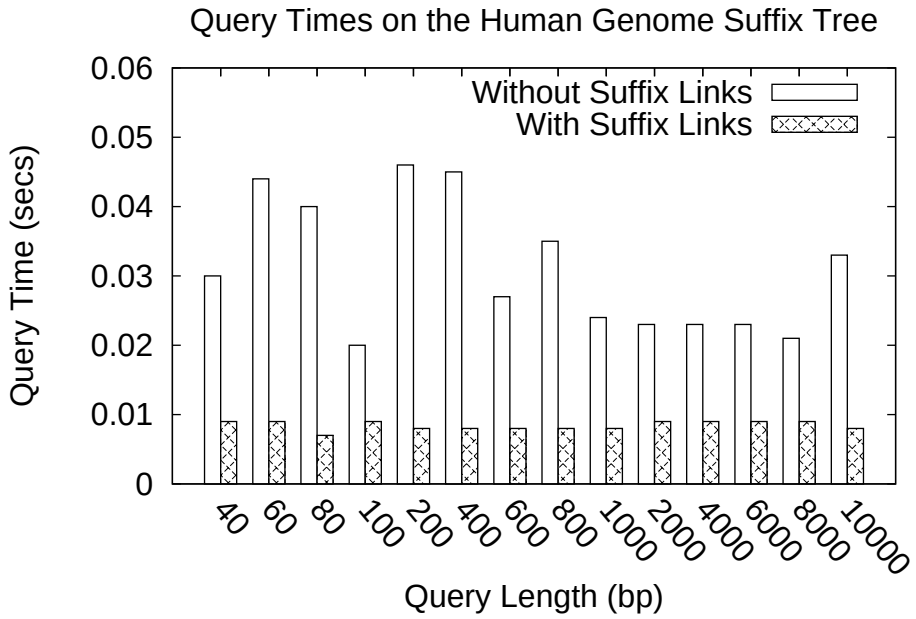


Figure 6.2: Average time on 500 consecutive queries, with and without suffix links, on the human genome suffix tree.

into the human genome sequence, for a query of length l be the query length, we issued 500 queries, Q_j (with $j \in [0, 500)$), where $Q_j = s_{i+j}s_{i+j+1}\ldots s_{i+j+l-1}$. We compared the query times for TRELLIS with and without the use of suffix links. For the `with suffix links` case, we employ both the suffix links and the skip/count technique for 500 queries as mentioned above. For the `without suffix links` case, each of the 500 queries is searched starting at the root of its respective prefixed suffix tree. However, it is important to note that in this case too, we do employ the skip/count method to avoid matching each query character-by-character; only the suffix links are not used. Figure 6.2 shows the average times over the 500 queries for various query lengths. We find that using suffix links results in a 2 - 5 times speedup over not using the links. Note that the average query time with suffix links is under 0.01s for even the longest query length. Finally, for comparison, if TDD were run on the same queries, TRELLIS would outperform it by a factor of 10-15, since TDD does not have suffix links, and as shown in Figure 6.1, TDD average query times are between 0.1-0.15s, whereas TRELLIS with suffix links takes on average 0.01s.

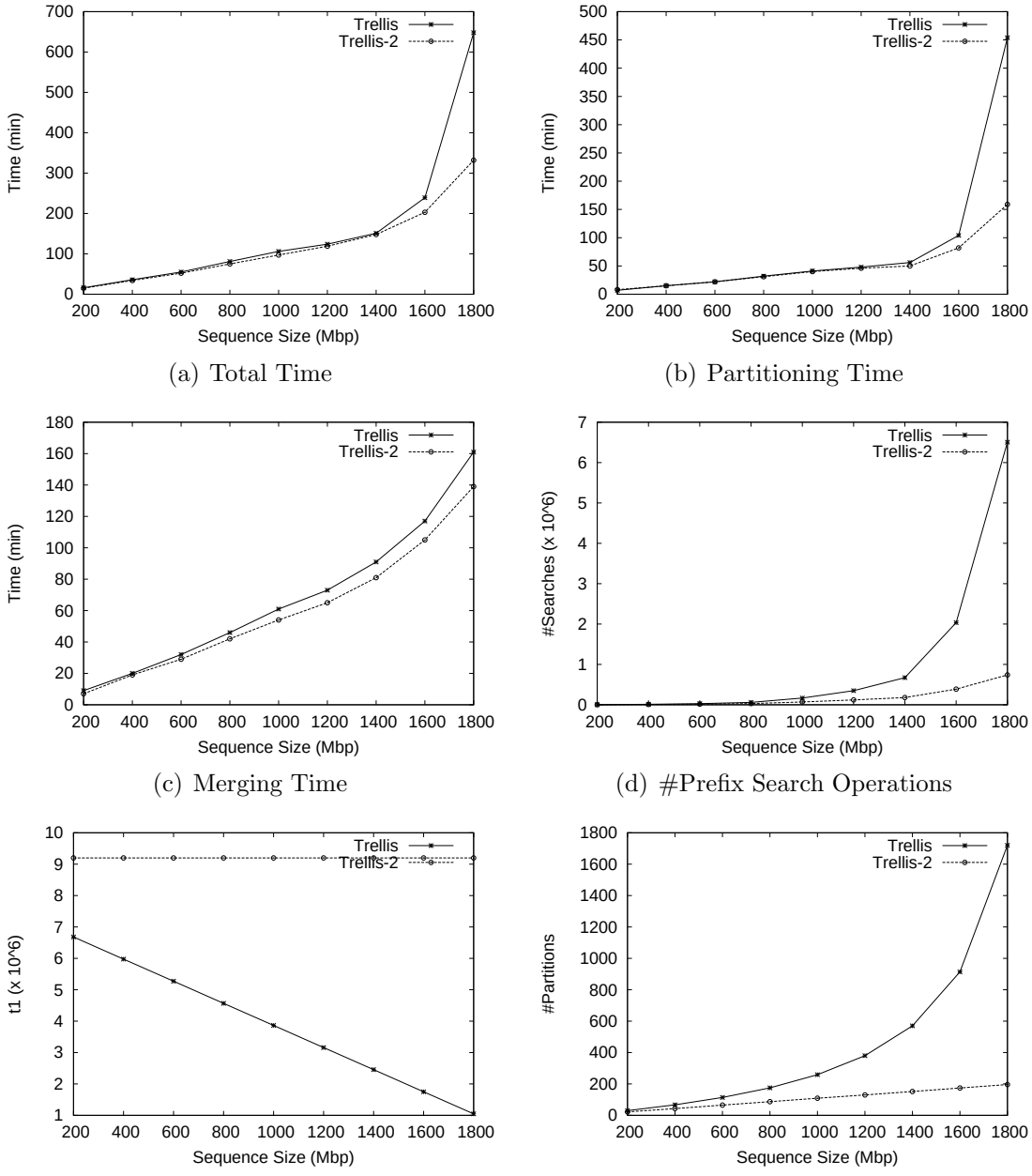
6.2 TRELLIS vs TRELLIS-2

In this section, we study the effects of the two optimization methods and compare TRELLIS-2 with the original TRELLIS. All experiments in this section were performed on a Power Mac G5 machine with 2.7GHz processor, 512KB of cache, 4GB of RAM, with 400GB disk space. The maximum amount of RAM usage across all experiments was restricted to 512MB. Note that the memory cap applies to all internal data structures including those for the suffix tree, memory buffers and the input string. Both TRELLIS and TRELLIS-2 were written in C++ and compiled with the GNU g++ compiler version 3.4.3 with optimization flags activated. The sequence data used in all experiments are segments of the human genome ranging from size 200Mbp to 2,400Mbp as well as the entire human genome.

6.2.1 Effects of Larger Partition Size: TRELLIS vs TRELLIS-2 (no buffer)

In this section, we study the effects of larger partition on this algorithm. Note that the buffer was not used for TRELLIS-2 in this set of experiments. The main benefit from the first optimization is that TRELLIS-2 has larger and therefore fewer partitions than TRELLIS. In TRELLIS the number of partitions is $O(\frac{n}{t})$ and the value of t decreases as n increases. Therefore, when indexing a very large sequence, TRELLIS is prone to a small value of t and consequently large number of partitions. In contrast, since the partitioning threshold t for TRELLIS-2 remains constant regardless of n, its number of partitions increases at a much slower rate. The results are shown in Figures 6.3(e) and 6.3(f).

The timings of TRELLIS-2 in comparison to TRELLIS are shown in Figures 6.3(a), 6.3(b), and 6.3(c). TRELLIS-2 consistently outperforms TRELLIS, especially when the input sequence size is much larger than the available memory. For example, TRELLIS-2 is about twice faster than TRELLIS for the 1.8Gbp input sequence. The reason for this improvement is two-fold. First, there are fewer but larger T_{R_i} in TRELLIS-2, which directly translates to fewer merging operations needed to construct the final suffix tree. Second, there are fewer T_{R_i,P_j} which incur file management overhead, such as disk seek operations and internal fragmentation, each time one is written to the disk (Figure 4.1 step c). During the disk write process, for



(a) Total Time

(b) Partitioning Time

(c) Merging Time

(d) #Prefix Search Operations

(e) Threshold for Partitioning Phase

(f) #Partitions

Figure 6.3: Optimization1: Effects of larger partition size. In this study, we compared the original TRELLIS with TRELLIS-2 (without buffer) and found that larger partition sizes resulting from the first optimization allow for faster suffix tree construction times.

each subtree T_{R_i}, each prefix P_j is searched in the subtree. If the prefix is found, all nodes below that prefix must be traversed and written to the disk as T_{R_i,P_j}. The process is repeated for all prefixes and for all partitions, therefore, the number of times T_{R_i,P_j} is written to the disk is exactly the number of prefix searches. Figure 6.3(d) shows the difference between the two algorithms. The number of prefix searches in TRELLIS increases at a much higher rate because of its dependency on the partition size. As a result, the partitioning phase time in TRELLIS increases at a similar rate. Since TRELLIS-2 does not suffer from the same behavior, it avoids much of the overhead and as a result outperforms TRELLIS.

6.2.2 Effects of String Buffer: TRELLIS-2 (without buffer) vs TRELLIS-2 (with buffer)

In this section, we investigate the effect of the string buffering strategy on TRELLIS-2 and show the comparison between TRELLIS-2 (without buffer) and TRELLIS-2 (with buffer). First we report the difference in merging time among several combinations of the buffering techniques in Figure . In this figure the data were gathered from indexing the first 200Mbp of the human genome with 512MB memory limit. The data confirms our finding on the buffer hit rates illustrated in Figure 6.4.

The rest of the experiments were conducted on the first 200, 400, . . . , 1800Mbp of the human genome for the no-buffer case. When using the string buffer, the sequence lengths tested were up to 2400Mbp. Experimental results in Figure 6.5(a) show that initially using no buffer outperforms using buffer. However, as the input sequence becomes much larger, the no-buffer TRELLIS-2 is left with less memory to construct the tree with because it has to maintain the entire compressed input string in memory. Consequently, it starts to outperform its buffered counterpart with deteriorating margin until reaching its memory restriction limit. Note that at 2Gbp, TRELLIS-2 without buffer had roughly 20MB left to construct the tree with. The timing worsened exponentially: it was running for more than 8 hours and it still had not finished the prefix creation phase, so we discontinued the run.

Figure 6.5(a) shows that, when using the string buffer, TRELLIS-2 gracefully

scales up to the input string size much larger than the available memory, i.e. $\frac{n}{4} \gg M$. To complete the experiments, we also ran TRELLIS-2 with buffer on the entire human genome. The program finished in approximately 11 hours using 512MB. To the best of our knowledge, a disk-based suffix tree construction of the human genome using only this small amount of memory has never been reported in literature before.

Unlike in the first optimization, TRELLIS-2 without and with buffer have different numbers of variable-length prefixes. Figure 6.5(e) shows the comparison between the two versions. The former is prone to a larger number of prefixes as the input string size increases because it maintains the entire input string in memory. The latter requires much fewer prefixes because it uses the string buffer which requires much less space. Note that when using string buffer, some memory must be deducted from the total available to maintain the small suffix tree of the $S[0 \ldots 2 \times 10^6)$ sequence. Therefore, there are slightly more partitions in the buffered version as shown in Figure 6.5(f). The subtrees T_{R_i} as a result are somewhat smaller when using the

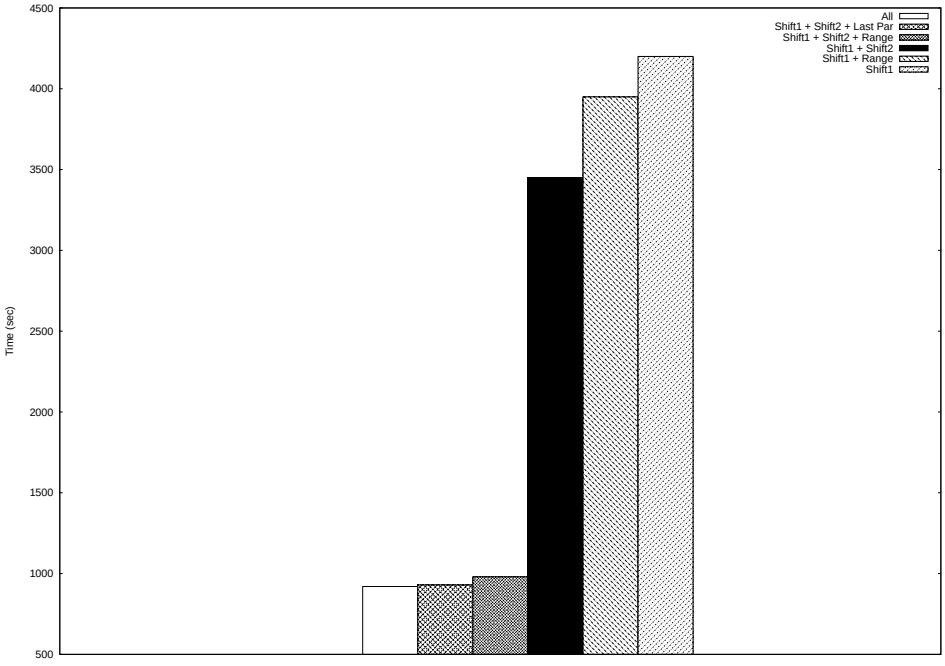
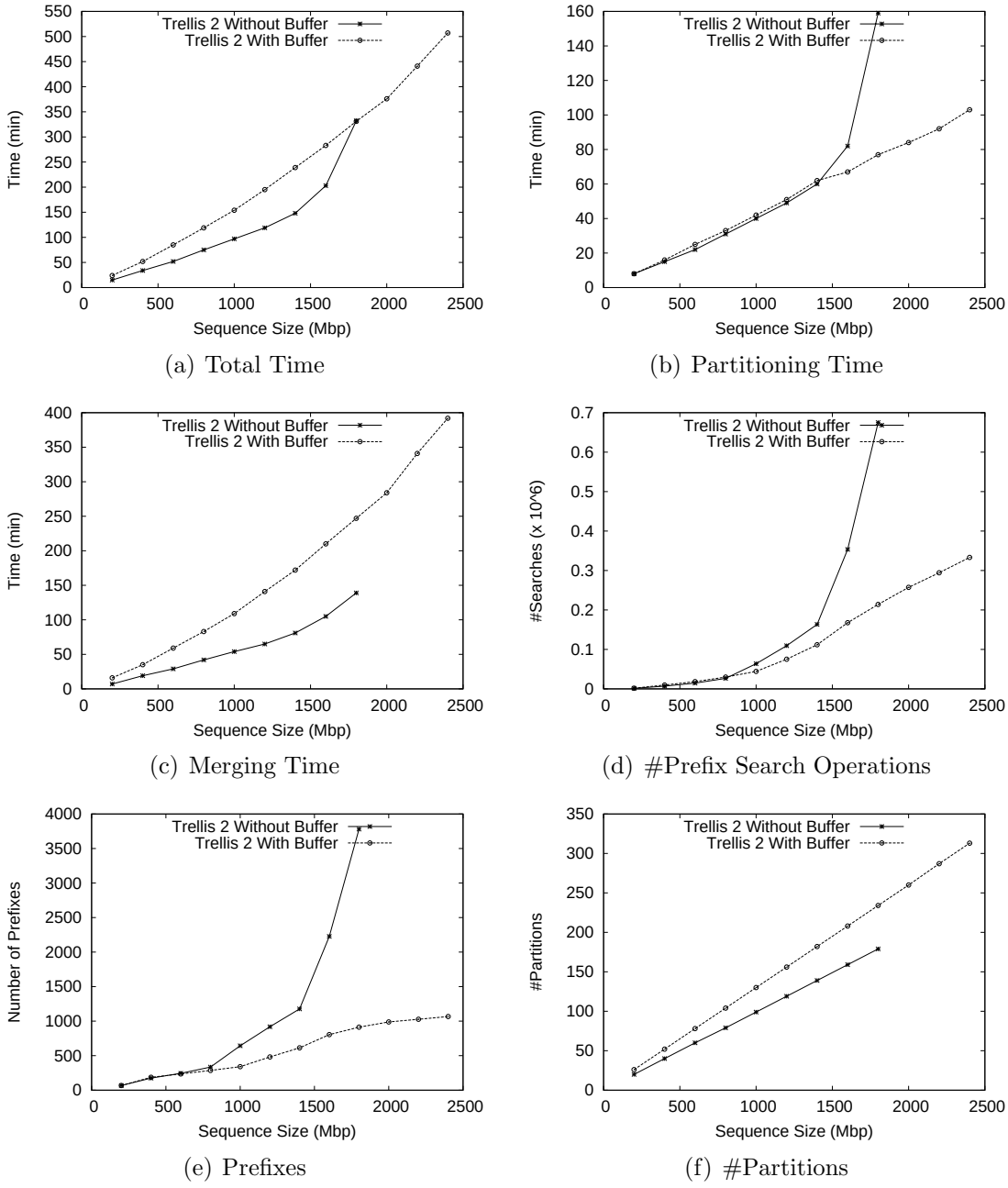


Figure 6.4: Merging time for different combinations of buffering techniques. The data were gathered from indexing the first 200Mbp of the human genome with 512MB of memory. The timings are consistent with the buffer hit rates displayed in Figure 5.6



(a) Total Time

(b) Partitioning Time

(c) Merging Time

(d) #Prefix Search Operations

(e) Prefixes

(f) #Partitions

Figure 6.5: Optimization2: Effects of the string buffer. In this study, we compared TRELLIS-2 (with buffer) with TRELLIS-2 (without buffer). The results show that using the buffer is slower than not using it due to the time spent in accessing the disk. However, the buffer makes TRELLIS-2 much more scalable.

buffer. Even though the difference is not as pronounced as in the first optimization, but we note it here for the sake of completeness.

Figure 6.5(d) shows the number of prefix searches required when writing the subtrees T_{R_i,P_j} to the disk. Similar to the result of the first optimization, the partitioning phase time is heavily influenced by the number of prefix searches. This is shown in Figure 6.5(b). As for the merging time shown in Figure 6.5(c), TRELLIS-2 with buffer performs worse than its no-buffer counterpart due to disk accesses incurred by the buffer misses, however the timings are within a reasonable time frame (external suffix tree construction typically takes hours at the genome level on a modern computer). In summary, the performance of TRELLIS-2 without buffer is good initially but degrades quickly as the input sequence becomes very large. In addition, the algorithm cannot continue once its memory restriction limit is reached. On the contrary, although using buffer slows down TRELLIS-2 due to disk accesses, it allows the algorithm to scale gracefully when the input string is much larger than the available memory. When deciding whether to use buffer or not, one should consult the amount of memory left after taking into account the entire input string. If too little is left, e.g. fewer than 100MB, then the buffer should be used.

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1 Conclusion

The amount of biological sequence data is growing exponentially. Recently, the International Nucleotide Sequence Database Collection (INSDC) announced that the DNA sequence database size has exceeded 100 Gbp [31]. To analyze such large amount of data, time- and space- efficient methods are necessary.

Suffix trees have been used extensively in a variety of string-based applications. In bioinformatics, the data structure has been adopted increasingly in the past decade to solve problems such as sequence comparison, motif discovery, and database search [24]. Many prior efforts have been made to develop practical methods for constructing suffix trees on very large sequential data, especially genome-scale sequences. However, a majority of these methods do not scale well for even moderately sized datasets [35]. A couple of existing methods do scale gracefully, however they do not provide suffix links required for efficient suffix tree search in many existing bioinformatics applications. Therefore, in such cases, these suffix tree construction methods would not be immediately applicable.

In this work, we developed Trellis, a time- and space- efficient disk-based suffix tree construction algorithm. Trellis builds the suffix tree based on a partitioning method via variable length prefixes and a suffix subtree merging algorithm. Trellis also provides a post-construction phase that very quickly recovers the full set of suffix links, should they be required by an algorithm using the disk-based suffix tree. Via the use of variable length prefixes, Trellis does not suffer from the data skew problem exhibited in many other disk-based suffix tree methods. In addition, since Trellis only works with a subtree that is guaranteed to fit entirely in the memory at one time, it bypasses the need to use a buffering scheme to manage suffix tree nodes during the tree construction, and as a result, avoids large disk I/O overhead. Trellis scales gracefully when the input DNA sequence is very large. We experimentally demonstrated that Trellis outperforms the state-of-the-art suffix

tree construction methods (for both with- and without- suffix links categories) by substantial margins. In addition, using only 2GB of memory, TRELLIS was able to construct the suffix tree of the entire human genome ($\approx$ 3Gbp) in 4.2 hours *and* recover all of its suffix links in 1.7 hours. The time taken by TRELLIS to construct the tree with and without suffix links are faster than the time taken by the best previous algorithm to just construct the tree by factors of 2 and 3, respectively. To the best of our knowledge, TRELLIS is the first algorithm that is able to effectively and efficiently achieve these challenging tasks. Besides its faster tree construction times, TRELLIS also exhibits superior query times; it outperforms the best current method by a factor of 2-3, when searching for queries directly on disk. In addition, our experiments show that having suffix links provides an additional 2-5 times speedup over not using the suffix links. All of the above factors make TRELLIS an attractive approach for storing and querying large sequence databases.

In addition to TRELLIS, we also proposed TRELLIS-2, a faster and more scalable version of our algorithm. Two optimization strategies are introduced. The first is based on a simple but effective observation that a larger partition size can be applied in the first step of the algorithm, thus reducing the number of merging operations required as well as disk I/O overhead. Experimental results show that, with this optimization, TRELLIS-2 consistently outperforms the original TRELLIS. The improvement is even more pronounced with input sequences that are much larger than the memory size. Specifically, when using 512MB of RAM, TRELLIS-2 is almost twice faster than TRELLIS at indexing 1.8Gbp of the human genome sequence. We also indexed the human genome with TRELLIS-2 using 2GB of memory, and the algorithm was able to finish in exactly 4 hours.

In the second optimization, we introduced a novel string buffering strategy for the tree merging phase. While TRELLIS requires the entire compressed input sequence to reside in memory, TRELLIS-2 does not fall under the same restriction because the string buffer requires much less memory than the entire string. The string buffer removes the memory limitation on the input string and, as a result, makes TRELLIS-2 much more scalable. Experimental results show that TRELLIS-2 with buffer scales gracefully to sequences much larger than memory. In particular,

it was able to index 2.4Gbp of the human genome in under 9 hours and the entire human genome in under 11 hours using just 512MB of RAM. To the best of our knowledge, the human genome suffix tree has not previously been reported to be constructed within this limited amount of memory before.

7.2 Future Work

As part of our future research, we plan to make Trellis more versatile by applying it to a wider range of alphabets, e.g., protein or English alphabets. Currently, Trellis's superior performance partly relies on the use of an array structure to keep track of children nodes. However, as the alphabet size increases, the constant overhead per node may cause the algorithm to become less efficient. In fact, a much previous implementation of Trellis was adapted to index successfully the entire SwissProt database [14] using under 1GB of main memory. In this case, each protein character was translated into one of the seven prediction values generated by HMMSTR [6] (In other words, the alphabet size was seven). Trellis was combined as a backend to the PSIST [19] algorithm to aid in its protein indexing. Much further studies are still required in order to optimize Trellis performance with different types of alphabets and/or sequence patterns.

Another goal is to create a user interface for Trellis. Most of the disk-based suffix tree construction algorithms published so far only report on the theoretical aspects of the suffix trees. We believe that, for the disk-based suffix tree to become practical and widely used in biological applications, a well-documented and user-friendly interface must be created.

We also plan to create a buffer to manage the suffix tree nodes during the query processing. Although it is important to have an efficient algorithm that constructs external suffix trees, it is equally critical to be able to use those trees efficiently. Buffer management policy should help speeding up the suffix tree reads for any algorithm using the tree. Currently, there is only one such buffer management algorithm, Stellar [2], from the authors of TOP-Q. Their buffer policy focuses on finding maximal exact matches between a query string and the reference string of the suffix tree. As part of our future work, we plan to create a buffer management

policy that targets a wider range of algorithms. The ideal buffer would be able to adjust itself automatically based on the node access pattern incurred by the algorithm using the disk-based suffix tree. Finally, we plan to parallelize TRELLIS which follows as a logical next step, since its partitioning and merging steps seem ideally suited to parallelization.

LITERATURE CITED

[1] 2006. DOGS – Database of Genome Sizes
`http://www.cbs.dtu.dk/databases/DOGS/`.

[2] S. Bedathur and J. Haritsa. Search-optimized suffix-tree storage for biological applications. In *IEEE Int'l Conf. on High Performance Computing*, 2005.

[3] S.J. Bedathur and J.R. Haritsa. Engineering a fast online persistent suffix tree construction. In *20th Int'l Conference on Data Engineering*, 2004.

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

[5] A.L. Brown. Constructing genome scale suffix trees. In *2nd Asia-Pacific Bioinformatics Conference*, 2004.

[6] Christopher Bystroff, David Baker, and Vesteinn Thorsson. HMMSTR: a hidden markov model for local sequence-structure correlations in proteins. *J. Mol. Biol.*, 301:173–190, 2000.

[7] A. Carvalho, A. Freitas, A. Oliveira, and M. Sagot. Efficient extraction of structured motifs using box-links. In *11th Conference on String Processing and Information Retrieval*, 2004.

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

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

[10] R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. In *14th Annual Symp. on Combinatorial Pattern Matching*, 2003.

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

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

[13] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. In *Workshop on Algorithm Engineering and Experiments*, 2005.

[14] ExPASy. Swiss-prot and trembl. `http://www.expasy.org/sprot/`, 2006.

[15] M. Farach-Colton. Optimal suffix tree construction with large alphabets. In *39th Annual Symposium on Foundations of Computer Science*, 1997.

[16] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *39th Annual Symp. on Foundations of Computer Science*, 1998.

[17] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[18] Paolo Ferragina and Roberto Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[19] Feng Gao and Mohammed Javeed Zaki. PSIST: Indexing protein structures using suffix trees. In *Proceedings of the IEEE CSB*, pages 212–222, 2005.

[20] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software Practice & Experience*, 33(11):1035–1049, 2003.

[21] Robert Giegerich and Stefan Kurtz. A comparison of imperative and purely functional suffix tree constructions. *Science of Computer Programming*, 25(2-3):187–218, 1995.

[22] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

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

[24] Dan Gusfield. Suffix trees (and relatives) come of age in bioinformatics. In *IEEE Computer Society Bioinformatics Conference*, 2002.

[25] K. Heumann and H. W. Mewes. The hashed position tree (HPT): A suffix tree variant for large data sets stored on slow mass storage devices. In *3rd South American Workshop on String Processing*, 1996.

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

[27] E. Hunt, M.P. Atkinson, and R.W. Irving. A database index to large biological sequences. In *27th Int'l Conference on Very Large Data Bases*, 2001.

[28] R. Japp. The top-compressed suffix tree: A disk-resident index for large seqeuences. In *Bioinformatics Workshop, 21st Annual British National Conference On Databases*, 2004.

[29] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software Practice & Experience*, 29(13):1149–1171, 1999.

[30] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[31] NCBI. Public collections of dna and rna sequence reach 100 gigabases. `http://www.nlm.nih.gov/news/press_releases/dna_rna_100_gig.html`, 2005.

[32] Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 833–844. ACM, 2007.

[33] K.-B. Schürmann and J. Stoye. Suffix tree construction and storage with limited main memory. Technical Report 2003-06, Universität Bielefeld, 2003.

[34] S. Tata, R.A. Hankins, and J.M. Patel. Practical suffix tree construction. In *30th Int'l Conference on Very Large Data Bases*, 2004.

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

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

[37] E. Ukkonen and J. Kärkkäinen. Sparse suffix trees. In *2nd Annual Int'l Conference on Computing and Combinatorics*, 1996.

[38] P. Weiner. Linear pattern matching algorithms. In *14th IEEE Symp. on Switching and Automata Theory*, 1973.