TRELLIS: An Effective Algorithm for Constructing Disk-based Suffix Trees with Suffix Links

Benjarath Phoophakdee Mohammed J. Zaki

22 February 2006

Abstract

The amount of genetic codes available to the public has just reached an important milestone of 100 *Giga*base pairs this past August. This includes a number of large and complete genome sequences, such as the Human genome. The size of the Human genome is about 3 Gigabase pairs. It is becoming more common now to have a DNA sequence of that scale as an input to biological database applications. Common tasks performed on such sequences include exact/approximate matching queries as well as repetitive structure finding. Suffix trees have been widely acknowledged as an indexing data structure that performs the above tasks efficiently. Their construction can be done quickly (in linear time) for small sequences. Unfortunately, it has been shown difficult to construct a suffix tree for large sequences due to its infamous memory bottleneck problem. Several efficient disk-based suffix tree algorithms have been proposed in recent years. Many of them assume that the input sequences are uniformly distributed, which does not apply for DNA sequences in nature. Some disregard an important suffix tree feature, called *suffix links*, required in many existing string-processing algorithms. Only two of the currently existing suffix tree methods do not exhibit these drawbacks, however, they have not been reported to scale to the Human genome level. In this work, we propose a new disk-based suffix tree algorithm with $O(n)$ complexity, called TRELLIS [1]. Our algorithm does not assume any specific distribution and also maintains a partial set of suffix links, which were experimentally shown to provide speedups over no suffix links at all. TRELLIS was able to index the Human genome with just under 1GB of memory. Our algorithm outperforms the other two algorithms mentioned above by orders of magnitude in construction time.

1 Introduction

Over the past several decades, a huge amount of DNA data has continuously been sequenced in laboratories around the world. The data growth rate has been exponential [32], and just in this past August, the public collection of DNA and RNA sequences has finally reached a very important milestone of 100 gigabase pairs [31]. As an interesting frame of reference, "one hundred billion bases is about equal to the number of nerve cells in a human brain and a bit less than the number of stars in the Milky way" [31]. These huge amounts of sequences are collected from genes and genomes from over 165,000 organisms. A single genome may be as large as billion basepairs. For example, the human, rat, and mouse genomes are approximately 3, 2.7, and 2.5 billion bases respectively. As a consequence of the data's enormous size and extreme growth rate, it is critical for researchers to have effective data structures and efficient algorithms for storing, querying, and analyzing these sequence data.

[1] TRELLIS is an anagram of the bold letters in the phrase: **E**xternal **S**uff**I**x **TR**ee with **L**inks for **L**ong Sequences

Suffix tree is an indexing structure widely acknowledged to perform the above tasks effectively. Its versatile data structure can be applied to solve very quickly a variety of string-based problems, such as exact and approximate matching, exact set matching, database querying, finding the longest common substrings of more than two strings, etc [23]. One of its most common applications in bioinformatics is exact match detection, where later the matches serve as seeds or anchors in an alignment program. The alignment programs that use exact matches as their anchors include MUMmer [11, 12, 28], EMAGEN [14], AVID [3], MAVID [4], and MGA [24].

Weiner [38], McCreight [29], and Ukkonen [36] are the three classic algorithms that construct suffix trees in linear time and space given that the trees fit entirely in the main memory. A variety of efficient in-memory suffix tree construction algorithms were also proposed later [21, 17, 16, 37, 18]. However, these algorithms are not designed to scale as efficiently when the input sequence is extremely large, due to the infamous *memory bottleneck problem*. As pointed out in [17], the random access to the input sequence is the main bottleneck. Additionally, traditional linear time algorithms rely on *suffix links*, which are a key feature in obtaining the linear time construction of the tree. While, during the tree construction, these algorithms traverse horizontally across the tree via the suffix links in order to quickly propagate the changes, they must traverse vertically down the tree to add new suffixes as well. Hence, a poor locality of reference which contributes to the memory bottleneck problem [25].

To address the issue, several disk-based suffix tree algorithms have been proposed in the last few years: [25, 5, 26, 33, 9, 1, 34]. In Hunt et al. [25], Japp [26], Schürmann et al. [33], and Tata et al. [34], the authors completely abandoned the use of suffix links and sacrificed the theoretically superior linear construction time in exchange for a better locality of reference. Unfortunately, some existing fast string-processing algorithms, such as finding tandem repeats in linear time [22], extracting structral motifs [7], computing matching statistics and approximate pattern matching [8], rely heavily on suffix links and therefore are inapplicable with these disk-based suffix trees. Furthermore, Hunt's [25], Japp's [26], Schürmann's [33], and Brown's [5] methods assume that the input sequences are uniformly distributed, which is not true for DNA sequences in nature. As a result, these algorithms exhibit the data skew problem, where some paritions of the suffix tree would *not* fit entirely in the main memory as expected. TOP-Q [1] and DynaCluster [9] are currently the only existing suffix tree algorithms without the data skew problem that also maintain the suffix links. However, neither one of them has reported to scale up to the Human genome level.

In this work, we present a novel approach to construct suffix trees on disk. Specifically, we made the following contributions:

1. We proposed a O(n) new disk-based suffix tree construction algorithm, called TRELLIS, based on a novel idea of constructing the tree by *partitioning and merging*.

2. TRELLIS introduces the use of *variable-length prefixes*, which solves completely the data skew problem exhibited in several other previous disk-based suffix tree construction algorithms.

3. TRELLIS scales gracefully for large DNA sequences. Specifically, it was able to index the Human genome in 36 hours using slightly less than 1GB of memory! To our knowledge, TDD is the only other algorithm reported to index successfully the Human genome, however TDD completely forgoes the suffix link structure. Therefore, TRELLIS is the first and currently only disk-based suffix tree construction algorithm, with partial suffix links, that is capable of handling an input DNA sequence this large.

4. TRELLIS maintains a partial set of suffix links. Although they are not the complete set, we

conducted extensive studies which show that they provide speedups in query matching over having no suffix links at all.

5. TRELLIS is compared to the only other existing algorithms that do not exhibit the data skew problem and also maintain the suffix link structure, TOP-Q [1] and DynaCluster [9]. These algorithms have reported to successfully construct, with suffix links, only chromosome-scale input sequences. Our algorithm is shown to be faster in construction time by orders of magnitude!

2 Related Work

Let us begin this section by stating that there are also other research work on external indexing structures, such as external suffix arrays [13] and string B-trees [19]. The focus of this work is only on external suffix trees.

The first disk-based suffix tree construction algorithm was introduced in Hunt et al. [25]. The authors abandoned the use of suffix links, which enable linear construction time, in exchange for a better locality of reference. The method first calculates a set of *fixed*-length prefixes of all the input string's suffixes based on the available memory, such that a suffix tree of all suffixes that begin with any given prefix can fit entirely in the main memory. Then for each fixed-length prefix, the method makes one pass over the string and inserts all suffixes starting with the given prefix into the disk-based suffix tree. Hunt's method is theoretically $O(n^2)$, where n is the size of the input string.

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 as in Hunt's. The main improvement is that insertions of suffixes do not start at the root of the suffix tree, but instead at the root of the subtrees, which is at the depth of the prefix length. The authors stated that their algorithm is suitable for the construction of large suffix trees as long as the memory size is six times as big as the input sequence length.

Top-compressed suffix tree algorithm was introduced in [26]. The author improved upon Hunt's algorithm by introducing a pre-processing stage and by using parallel index construction. Top-compressed suffix trees use a similar pre-partitioning method to Hunt's and assume that the partitions are equal in size.

Hunt's static pre-partitioning method via fixed-length prefixes exhibits a few drawbacks as discussed in detail in [9]. One of them was the difficulty in handling data skew. Since the characters in real DNA sequences are not uniformly distributed, some partitions may fit in the memory while some may not. Cheung et al. [9] addresses this issue via dynamic clustering.

In DynaCluster [9], the suffix tree is created one cluster (a cluster represents a group of nearby nodes) at a time. The clusters provide a small locality, and therefore a large disk-based suffix tree can be built using a limited size of memory. DynaCluster also drops the use of suffix links during the tree construction to allow a better locality of reference and is shown experimentally to be $O(nlgn)$. An optional phase that rebuilds the suffix links after the entire tree is constructed on disk is provided. DynaCluster is one of the only two existing algorithms that do not exhibit the data skew problem as well as maintains the suffix links. Its construction time is reported to be much faster than its suffix link rebuild time. The main cost of the suffix link rebuild is from its preparation phase, which requires two full traversals of the complete disk-based suffix tree.

Recently, Bedathur et al [1] developed the TOP-Q buffer management policy for online disk-based suffix tree construction. The TOP-Q policy takes into account the probabilistic behavior of

the suffix tree construction and retains the nodes that are more likely to be accessed during the construction time in the buffer and evicts the rest. TOP-Q was incorporated with the Ukkonen algorithm and resulted in a disk-based suffix tree construction algorithm that did not sacrifice the suffix links. Since TOP-Q is based on the Ukkonen method, it is a linear time algorithm. Additionally, TOP-Q does not assume any specific data distribution. TOP-Q and DynaCluster are currently the only algorithms that maintain suffix links as well as do not exhibit the data skew problem. However, they have not been reported experimentally to scale up to the Human genome level.

Another $O(n)$ suffix tree algorithm was introduced in [5]. The author improved upon Hunt's algorithm by optimizing the insertion of suffixes into partitioned suffix trees 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 also suffers from the data skew problem.

A disk-based suffix tree construction method, called TDD, was described in [34]. Similar to Hunt's approach, TDD drops the use of suffix links in exchange for a much better locality of references. It uses a Partition and Write Only Top Down method, which is based on the wotdeager algorithm [21], combined with a specialized buffering strategy that allows better cache usage during tree construction. Although TDD is an $O(n^2)$ algorithm, it was experimentally shown to outperform both Hunt's approach and TOP-Q. TDD is currently the only algorithm reported to scale up to the Human genome level.

Lastly, as an improvement to TDD, a new disk-based suffix tree construction algorithm based on a sort-merge paradigm was introduced in [35]. 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 new algorithm 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. Although this algorithm is more efficient than TDD, it suffers the same drawback, which is the nonexistence of suffix links.

As an alternative to the above disk-based approaches, a linear time in-memory distributed suffix tree algorithm was proposed in [10]. The author adapts Ukkonen's algorithm and constructs large-scale suffix trees using a cluster of computing nodes, where at each node a partition of suffix trees is computed. The notion of partitions used here is similar to that in Hunt's. The notion of sparse suffix links is introduced here, where they denote the suffix links existing strictly within any given partition (as opposed to in our algorithm, where the suffix links exist both within and across the partitions). The sparse suffix links are used to enable the linear time construction of the partitions.

Our approach constructs the suffix trees by a novel partitioning and merging method based on the Ukkonen's algorithm. Our goals are to maintain the suffix links and avoid the data skew problem without any complicated buffer management policy, as well as to be able to gracefully scale to the Human genome level.

3 TRELLIS ALGORITHM

In this chapter, we present a novel disk-based suffix tree construction algorithm called Trellis. Trellis is a three-phased algorithm. The first phase computes variable-length prefixes based on the input string. The second phase partitions the input string into many small substrings, creates a suffix tree for each partition, and stores them in form of prefixed suffix sub-trees. The third phase merges the prefixed suffix sub-trees of the same prefixed together, and stored the merged prefixed suffix trees on disk with respect to their prefixes. An overview of Trellis is shown in Fig. 1.

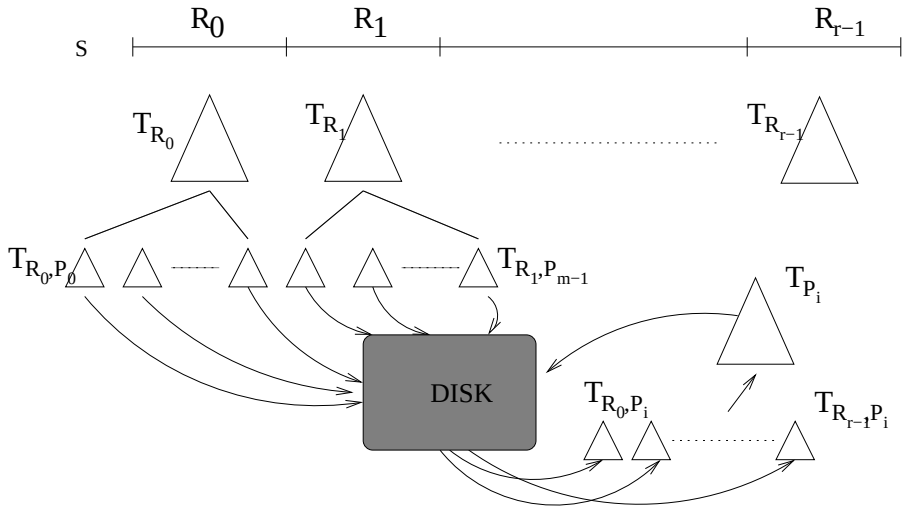

Figure 1: TRELLIS: Overview of the algorithm. The notions used here are defined in the Preliminaries section.

3.1 Preliminaries

This section introduces suffix tree notions to aid the presentation of TRELLIS. Most of the notions listed here are commonly used, however some are specific to our algorithm.

Let Σ denote a set of characters (or the alphabet), and let $|\Sigma|$ denote the size of the set. 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 $|S| = n$. The i^{th} suffix of S is represented as S_i, where $S_i = s_i s_{i+1} s_{i+2} \ldots s_{n-1}$. For convenience, we append the terminal character to the string, and refer to it by s_n.

The suffix tree of the string S, denoted as T_S, stores all the suffixes of S in a tree structure, where suffixes that share a common prefix lie on the 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, must have at least 2 children, where each edge of its children begins with a different character. Since the terminal character is unique, each suffix differs from another in at least one position. Thus, there as 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 sum of the edge lengths on the path from the root to v. Let $\sigma(v)$ denote the substring obtained by traversing from the root to v, then $|\sigma(v)| = d(v)$. 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_S for a sample DNA sequence, $S = ACGACG\$$ is shown in Fig. 2.

Some additional notations are introduced here to describe our disk-based suffix tree. TRELLIS partitions the input string into several equal-length substrings (with an exception of the last one, which maybe shorter than the others), and builds a suffix tree for each partition. Let t be a threshold value denoting the partition size, such that a suffix tree of a size-t string can be created and reside entirely in the memory. Since the number of nodes in a suffix tree is bounded by $2n$ [27], where n denotes the input string size, t can be any number that respects the equation $(2t \times \#bytes\_per\_node) \leq A_m$, where A_m is the available memory. Given the threshold t, the input string is then split into $r = \lceil \frac{n+1}{t} \rceil$ partitions of size at most t. Let R_i refer to partition i, with

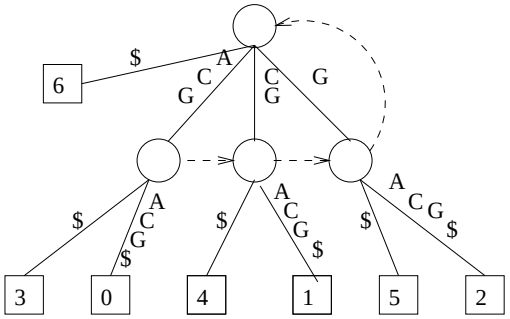

Figure 2: Suffix tree T_S for string S = "ACGACG$". The circles represent internal nodes, and the squares represent leaf nodes. The leaf nodes are numbered with respect to their corresponding suffixes

$0 \leq i \leq r-1$.

Let $P = P_0, P_1, P_2, \ldots, P_{m-1}$ denote the set of *variable-length prefixes* (to be formalized later) of S. The threshold t is also used to determined the set of variable length prefixes P, such that the frequency of any prefix P_i, where $0 \leq i \leq m-1$, occurring in S is no greater than t. Let *sup(P_i)* denotes the support, i.e. frequency, of the prefix P_i.

Let T_{R_i} denote the suffix tree of the i^{th} partition (R_i) of S, and let T_{R_i,P_j} denote the prefixed suffix sub-tree that consists of all suffixes from R_i that begin with the prefix P_j. Finally, let T_{P_i}, with $0 \leq i \leq m-1$, denote the prefixed suffix tree that consists of all suffixes of S that begin with P_i.

An example to demonstrate the prefixed suffix trees of prefixes AC, CG, and GA is shown in Fig. 3. For the sake of simplicity, here we let the prefix length equals to two. In general, the prefixes maybe of unequal length. In this example, the suffix $G\$$ (S_5) does not have its own prefixed suffix tree because S_5 is a prefix by itself and there is no string left to build a tree on.

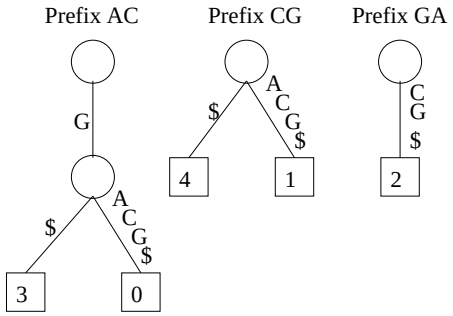

Figure 3: Prefixed suffix trees of S for prefixes of length = 2: AC, CG, and GA

3.2 Suffix Tree Partitioning

3.2.1 Traditional Fixed-length Prefix Approach and its Shortcomings

The idea of partitioning a suffix tree based on the prefixes of each suffix was originally introduced in [25]. Their method partitions the suffix tree based on fixed-length prefixes. Two variations of

this approach were outlined. The first is to scan the input sequence once, count the number of occurrences of each fixed-length letter pattern, and then use a bin-packing algorithm to pack each partition with different prefixes. This approach does not assume that the input DNA sequence has the pseudo-random nature. Therefore, while some prefixes may require exactly the memory space of one partition, others may require less and the suffixes of such prefixes can be stored together in only one partition. The second approach assumes the DNA sequence possesses the pseudo-random nature. A sufficient length l is chosen such that, for each prefix of length l, its suffixes can be stored in the same suffix tree and the tree would fit entirely in the main memory. The suffix tree of each prefix is also called a partition. The fixed-length prefix approach was also adopted in other disk-based suffix tree algorithms, i.e. [26] and [5].

The problems of the fixed-length prefix approach are three-fold. Firstly, it is difficult to handle data skew due to the fact that DNA sequences in nature are not uniformly distributed. For example, we gathered the following statistics on the Human genome (size is approximately 3GB). The frequencies of length-1 prefixes, i.e. A, C, G, and T, are about 30%, 20%, 20%, and 30% respectively. The frequencies of length-2 and length-3 prefixes are shown in Fig. 4 and Fig. 5. This data show that using fixed-length prefixes would not yield equal or even close partition sizes. Additionally, [9] demonstrates in details that using fixed-length prefixes results in a great portion of partitions being larger than expected. Since these partitions do not fit in the memory, they require tree node buffering which induces I/O cost. While increasing prefix length may present a quick fix that enables every partition to fit entirely in the memory, it would result in several partitions being smaller than necessary, and hence a waste of resources.

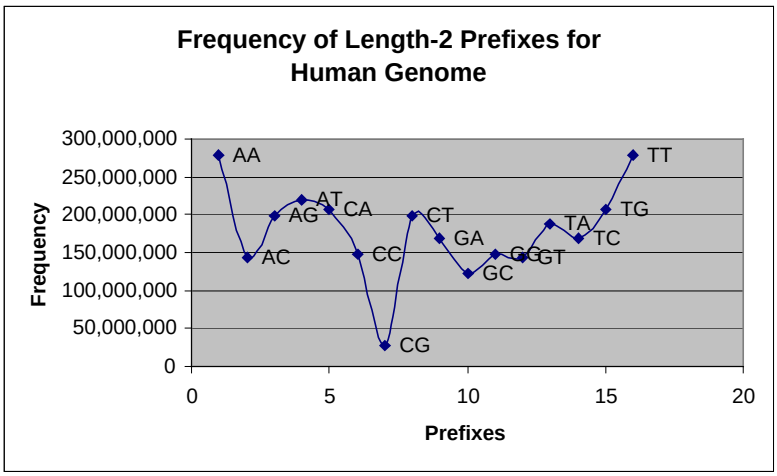

Figure 4: Frequency of Length-2 Prefixes for Human Genome

Secondly, even if the uniform distribution is assumed, the computation to count the number of occurrences of each fixed-length prefix will be very costly when the input DNA sequence is extremely long. Since the target sequences for disk-based suffix tree algorithms are for very long sequences, the fix-length approach will not be so applicable. For example, we conducted our experiment such that each partition can hold up to 3 million suffixes. In order to obey the limit, the prefix length needed was as high as ten. If we followed their second approach, we would have as many as 4^{10} or 1,048,576 partitions! Even worse, many of these partitions are small enough to reside together in the main memory but would be created separately, which is an unnecessary waste of resources. Our result shows that 47 out of 256 length-4 prefixes would result in sufficiently small partitions

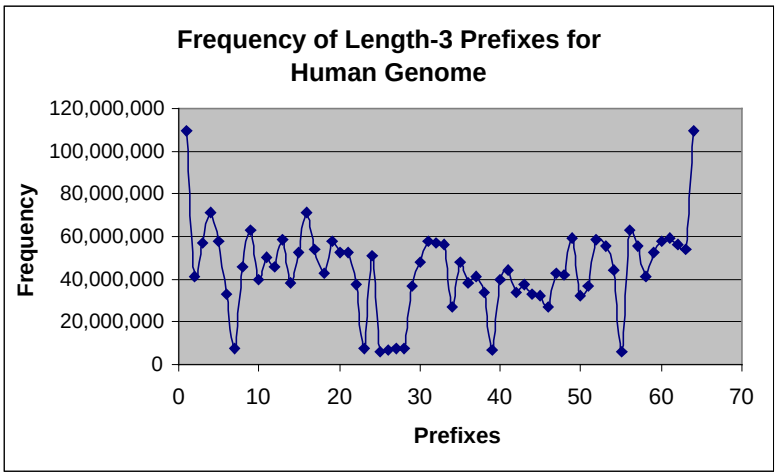

Figure 5: Frequency of Length-3 Prefixes for Human Genome. The data points represent the prefixes AAA, AAC, AAG, ..., TTG, TTT, ranging from left to right.

already. If we were to extend these prefixes up to length ten, the number of partitions created for them would be 47×4^6 or 192,512 partitions! Another problem is the size of the data structure that stores the fix-length prefixes. This data structure is of size $O(4^l)$, which can be a very big number. For example, the sizes are 4,194,304 and 16,777,216 already when l is 11 and 12. Also, it may not be very applicable when Σ is larger than 4, e.g. for protein sequences or English text.

Lastly, since the DNA sequence is not uniformly distributed, the value of l must be estimated. In [25], l was set to 3 and their experiments were done only at the chromosome level. However, when the input sequence is as large as the Human genome, l must be a bigger number. To make sure that every partition can reside in the memory, a high value can be chosen, e.g. 10. However, the space complexity will be high as explained above as well as some partitions would be unnecessarily small. On another hand, if the value chosen for l is not high enough, some partitions may not fit entirely in the memory and a buffering scheme would be required.

3.2.2 Variable-length Prefixes

In this section we present our solution to the all of the above problems. We observed that, while some prefixes need to be of a longer length in order for their partitions to fit in the main memory, other prefixes can be of a shorter length. TRELLIS applies this observation into solving such problems by extending the prefixes only as needed, hence *variable-length prefixes*.

A naive implementation of the approach would be as follows. First we gather the frequencies of all length-1 prefixes, extend only the prefixes whose frequencies are greater than the threshold t for another character, and continue extending in this manner until all prefixes of the current length pass the threshold. For example, suppose $sup(A) > t$, then we must extend A to AA, AC, AG, and AT, and find their supports. However, if $sup(A) \leq t$, then A becomes a valid prefix and need not be extended. The drawback of this approach is that the input sequence requires scanning each time a prefix is extended. A simple solution is to scan the input sequence once, and keep track of the frequencies of all prefixes from length 1 up to some length l such that all prefixes of length l have supports $\leq t$. Using the support information gathered from this step, we can start extending and creating the variable-length prefixes as described above. Although this technique would solve

the data skew problem, it has a major drawback such that we would have to guess for a sufficient value of t.

With some further investigations, the following statistics in Table 1 were collected from the Human genome. It can be observed that, as the prefix length increases, the number of prefixes with supports $> t$ sharply decreases. In fact, there are only two of such prefixes starting at the length equals to eight. We also discovered that when the prefix length increases, the only prefixes most likely to have the supports greater than t are precisely the ones composed of all A's and all T's. They are also the two prefixes that do not pass the threshold here. This behavior can be observed from the graphs in Fig. 4 and Fig. 5 above as well.

Prefix Length	#Possible Prefixes	#Prefixes with Support $> 1,000,000$
1	4	4
2	16	16
3	64	64
4	256	253
5	1,024	781
6	4,096	930
7	16,384	68
8	65,536	2
9	262,144	2
10	1,048,576	2
11	4,194,304	2
12	16,777,216	2
13	67,108,864	2
14	268,435,456	2
15	1,073,741,824	2
16	about 3Gb	0

Table 1: The number of prefixes with support $> t$, where t is 1 million, from prefix length 1 - 16. When t equals to 2 and 3 million, the results are also similar. Note that since $4^{16} = 4,294,967,296$, which is greater than the size of the Human genome, the possible number of prefixes of length 16 is only the size of the Human genome minus 15.

The lesson learned here is that we can gather the frequencies of all prefixes starting from length 1 to 8 in one scan, then perform the prefix extension step. Based on the left-over length-8 prefixes that do not pass the threshold t, we continue collecting the frequencies of longer prefixes (from length 9 on) in another scan. We repeat these procedures until all prefixes pass the threshold. The benefit of doing multiple scans are:

- At length 9, instead of detecting 262,144 prefixes and storing their frequencies during a scan, we only need to do so for 8 prefixes. A length 10, instead doing the same for 1,048,576 prefixes, we only need to do so for 32, and so on.
- We do not have to *guess* the last value of prefix length because we repeat the procedures until all extended prefixes have supports $\leq t$.

There are a couple of questions here that need to be addressed.

1. How do we know that we should collect the frequencies up to length 8 during in the first scan?

2. When extending from length 9, to what length do we extend up to?

To answer these questions, we would like to emphasize that the goal here is to avoid unnecessary book-keeping of prefix frequencies. As shown in Table 1, when the prefix length increases, the number of possible prefixes at that length exponentially increases, while the number of prefixes that would actually require an extension sharply drops. Therefore, a possible improvement would be to update the prefix frequencies in intervals of prefix lengths.

During the first scan, we process the prefixes up to the length as large as possible in order to save on the number of scans, given that the data structure needed to store them is of a reasonable size. It is very important for the size of this data structure to be sufficiently small, so that 1) the frequency updates can be performed quickly and 2) not too many unnecessary updates are possible. Let L_i be the highest prefix length after the i^{th} scan. The size of $L_1 = 8$ presents a reasonable choice because $\sum_{j=1}^{8} 4^j$ or 87,380 is not too large and that was how we chose the value 8. (We could very well use 7 or 9.) Since we know that the distributions of the nucleotides are not uniform, it is likely that some of the length-L_1 prefixes will pass the threshold t and some will not. During the next scan, we only extend the length-L_1 prefixes that did not pass.

Let EP_1 be the set of prefixes that require further extension after the first scan. We choose the value of L_2 such that the resulting size of data structure to store the prefixes of length $L_1 + 1$ to L_2 or $\sum_{j=1}^{L_2 - L_1} EP_1 \times |\Sigma|^j$ is reasonably small. To answer the second question, in this case, EP_1 is 2 and we chose L_2 to be 12 because or $\sum_{j=1}^{4} 2 \times 4^j$ is only 360. We continued in this manner and let L_3 to be 16, and that was precisely where all variable-length prefixes obtained pass the threshold t. The resulting set of variable-length prefixes of the human genome when t equals to 1 million ranges from prefix length 4 to 16 and there are 6400 prefixes in total.

By computing the variable-length prefixes as described in this section, we completely avoid all of the problems established by using fixed-length prefixes.

3.2.3 Time and Space Complexity

The time required for finding variable-length prefixes can be computed as follows. Let l be the maximum prefix length. At each index of S, l prefixes must be checked to see if they are the prefixes being extended. If so, their frequencies must be incremented by one. Such updates require looking up the corresponding prefix frequencies, which takes $O(lg(|\Sigma|^l))$. Therefore, the total runtime of this step can be written as $O(|S| \times l \times O(lg(|\Sigma|^l))$. Since our $|\Sigma|$ is 4, the runtime can be reduced to $O(|S| \times l)$. Since $l \ll |S|$, the runtime is $O(|S|)$.

The space required for this step is bounded by the size of the data structure used to hold the frequencies of the prefixes during each scan. A loose bound would be $O(\sum_{j=1}^{l} |\Sigma|^j)$. However, since we extend the prefixes in intervals, a tighter bound would be $O(\sum_{i=1}^{maxScan}(\sum_{j=1}^{l_{max,i}-l_{min,i}+1} |EP_{i-1}| \times |\Sigma|^j))$, where $maxScan$ denotes the total number of input string scans, $l_{max,i}$ denotes the maximum prefix length during the i^{th} scan, $l_{min,i}$ denotes the minimum prefix length during the i^{th} scan, EP_i denotes the set of prefixes that require extension after the i^{th} scan, and lastly let $|EP_0|$ equals to 1.

3.3 Prefixed Suffix Sub-Trees

TRELLIS divides the input string S into $r = \lceil \frac{n+1}{t} \rceil$ substrings. Using the terminology defined in the Preliminaries section, the main idea of this phase of the algorithm is to create a suffix tree T_{R_i} for each i^{th} partition, such that T_{R_i} contains precisely all the suffixes of S from the i^{th} partition. Let's call the trees T_{R_i} *suffix sub-trees* because each one represents a portion of the complete suffix

tree of S. We adopt the Ukkonen's algorithm [36] for creating the suffix sub-trees because of two reasons: the efficient $O(t)$ construction time and the availability of suffix links.

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 variable-length prefixes, i.e. each subtree of T_{R_i} contains the suffixes of S that begins with the same prefix. Let's call these subtrees *prefixed suffix sub-trees*. In another words, each T_{R_i} is first separated into T_{R_i,P_j}'s, where $0 \leq j < m$, and each of the T_{R_i,P_j}'s is written onto separate files. Therefore, by the end of each i^{th} partition, at most m files will be created where each file represents a T_{R_i,P_j}. See Fig. 1 for an illustration of this step.

When a *prefixed suffix sub-tree* T_{R_i,P_j} is being stored, the tree is traversed in a depth-first manner. When a node is encountered, its ID, starting and ending indexes, levels from the root, and for internal nodes, the ID of the node that its suffix link points to, are written to the disk via a binary file. The amounts of space needed to store an internal and a leaf node here are 20 and 16 bytes respectively. The purpose of storing the *prefixed suffix sub-trees* in this manner is so that they can be rebuilt in the memory later during the next phase.

An important point must be made here regarding the suffix links. In traditional Ukkonen's algorithm, nodes pointed to by suffix links are accessed via pointers. However, when storing the suffix trees on disk, the pointer locations are no longer valid. We remedied the situation by storing unique IDs of nodes that the suffix links point to instead of storing the nodes' memory locations. As a result, we need to be able to retrieve a node's on-disk location given its ID. We explain how to do this in the next section.

3.3.1 Time and Space Complexity

In this phase, TRELLIS uses the Ukkonen's algorithm [36] to build the suffix tree and stores the tree on disk in the depth-first manner. The Ukkonen's method is $O(n)$ and traversing the tree in the DFS manner is also $O(n)$. Therefore, the time for this phase is $O(n)$.

The amount of memory required during this phase equals to the amount of memory needed for the suffix tree construction of a string of size t plus the memory to store the substring of each partition, i.e. $O(t)$.

3.4 Suffix Trees Merging

In this phase, the prefixed suffix sub-trees of the same prefix are loaded into the memory, merged together to form a prefixed suffix tree of that prefix, and then stored back to the disk. Specifically, for a prefix P_j, T_{R_0,P_j} and T_{R_1,P_j} are first loaded into the memory, and merged together, which creates the current merged tree. Next T_{R_2,P_j} is loaded into the memory, merged with the current merged tree, and so on. We continue in this manner for all R_i such that $0 \leq i < r$. The last merged tree is exactly T_{P_j}, which is stored back onto the disk. The algorithm proceeds until all P_j's, $0 \leq j < m$, are processed.

We developed a $O(n)$ time merging algorithm that merges two suffix trees together. The pseudocode is given in Fig. 6. We outline the merging algorithm below. When merging two suffix sub-trees together, we simultaneously traverse the trees in a depth first manner. Starting from the two root nodes, we merge the edges leading to their children such that the merging edges always start with the same characters. Four possible cases may occur when merging two edges together: 1) the labels partly overlap, e.g. $AGCAT$ and $AGCGT$, 2) the labels are exactly the same string, 3) the first label is the second label plus some additional characters, e.g. $AGCAT$ and AGC, and 4) the second label is the first label plus some additional characters, e.g. $AGCAT$ and $AGCATTA$.

In each case, we merge the overlapping parts of the labels together, update the original labels with the left-over strings, and continue merging the edges of the updated nodes together if needed.

For example, let edge $e1(p1, n1)$ and $e2(p2, n2)$ be the two edges to be merged. $p1$, $n1$, $p2$, and $n2$ are the nodes in the suffix sub-trees 1 and 2 respectively. Let $AGCAT$ and AGC be the labels of $e1$ and $e2$ in that order, and thus case 3 applies here. First we merge the overlapping part of $e1$ and $e2$ together, i.e. ACG, and assign this string as a label to a new node $n3$. We then update $n1$'s label with the left-over string AT. Next we set $p1$ as $n3$'s parent, and $n3$ as $n1$'s parent. Then we check if $n2$ has any child starting with $n1$'s now leading character, A. If so, we merge such edge with the edge from $n3$ to $n1$. Otherwise, we simply add $n2$'s children as $n3$'s children.

```
Input          : Suffix trees T1 and T2
Output         : Updated merged suffix tree T1
MergeTree (T1, T2):
Node r1 = T1's root;
Node r2 = T2's root;
foreach character c ∈ Σ do
    if r2 has a child that begins with c and r1 does not then
        Add that child to r1;
    else if r1 and r2 both have a child that begins with c then
        MergeNode(r1's child, r2's child);

Input          : Nodes n1 and n2
Output         : Updated merged node n1
MergeNode (n1, n2):
S1[b1...e1] = string on the incoming edge to n1;
S2[b2...e2] = string on the incoming edge to n2;
if S1[b1...i] == S2[b2...j], where i < e1 and j < e2 then
    Let n be a new node with a string label S1[b1...i];
    Reset the string label of n1 to S1[i + 1...e1];
    Reset the string label of n2 to S2[j + 1...e2];
    Remove n1 from its parent node;
    Add n as a child of n1's parent node;
    Add n1 and n2 as children of n ;
else if S1[b1...e1] == S2[b2...e2] then
    foreach character c ∈ Σ do
        if n2 has a child that begins with c and n1 does not then
            Add that child to n1;
        else if n1 and n2 both have a child that begins with c then
            MergeNode(n1's child, n2's child);
else if S1[b1...i] == S2[b2...e2], where i < e1 then
    Let n be a new node with a string label S1[b1...i];
    Reset the string label of n1 to S1[i + 1...e1];
    Remove n1 from its parent node;
    Add n1 as a child of n;
    foreach child C of n2 do
        if C begins with S1[i + 1] then
            MergeNode(n1, C);
        else
            Add C as a child of n;
else if S1[b1...e1] == S2[b2...j] where j < e2 then
    Reset the string label of n2 to S2[j + 1...e2];
    if n1 does not have a child that begins with S2[j + 1] then
        Add n2 as a child of n1;
    else
        MergeNode(n1's child that begins with S2[j + 1], n2);
```

Figure 6: MergeTree Algorithm

3.4.1 The Pattern of S Accessed during the Merge

The trees are merged based on the comparisons of edge labels. The input string S is incrementally accessed as prefixed suffix sub-trees from more partitions are being merged. For an example, given a prefix P_j, when T_{R_0,P_j} and T_{R_1,P_j} are being merged, the partitions R_0 and R_1 of S would be accessed. The merged tree contains now the suffixes from partitions R_0 and R_1 that begin with the prefix P_j. Then T_{R_2,P_j} is loaded into the memory and merged with the current merged tree of R_0 and R_1. The resulting merged tree now contains the suffixes from partitions R_0, R_1, and R_2 that begin with the prefix P_j, and so on. Notice that during any i^{th} merge, the substrings from R_0 to R_i would be accessed. We further investigate below and found an interesting access pattern of S.

The string label of any tree edge is denoted by a pair of starting and ending indexes into S. When 2 edges from different trees are being merged, if their string labels contain the same prefix string, a new internal node would be created so that the un-overlapping parts of the two labels can branch off of it. The new internal node needs to be assigned a pair of starting and ending indexes. An important point to emphasize here is that these indexes must always be from the partition with the *lower* partition number. For example, suppose the two merging edges are from R_0 and R_1 and their string labels are $ACCTA$ and $ACCAG$. The first label corresponds to index positions [10, 14] and the second label corresponds to index positions [50, 54]. This merging requires an internal node to be created. The new internal node would have the string label ACC with two children nodes branching off of it. The new node would be assigned [10, 12] (as opposed to [50, 52]) as its starting and ending indexes. The algorithm consistently assigns the new index positions to any new internal node created during the merge in this manner.

Since all partitions of S need to be accessed during the merge, it is desirable that S is kept in the main memory for fast access. However, it may become impractical do so when S is a very large string. For example, the Human genome would consume about 3GB of space. A possible solution is to create a string buffer to manage S.

With that goal in mind, we further investigate how S is accessed during the merge. In this case S is the Human genome. The data that show the locations of S that were accessed during the 100^{th} iteration of suffix sub-trees merging for prefix $AACG$ is displayed in Fig. 7. The graph exhibits the following interesting characteristics:

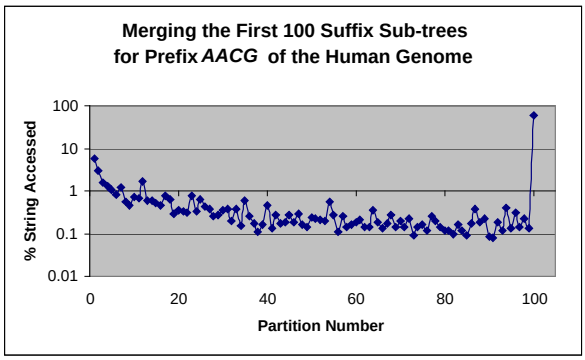

Figure 7: Percentage of S being accessed during the 100^{th} iteration of suffix sub-trees merging for prefix $AACG$

1. The partition accessed most is the *last* partition, i.e. R_{99}.

2. R_0 is accessed more than R_1, R_1 is accessed more than R_2, R_2 is accessed more than R_3, and so on.

Note that the access patterns during other partitions are also similar to what is shown here. The above observations can be explained as follows. During the 100^{th} iteration, $T_{R_{99},AACG}$ is loaded into the memory to merge with the tree created from merging $T_{R_0,AACG}$ to $T_{R_{98},AACG}$. The edge labels of $T_{R_{99},AACG}$ need to be compared during the merge, hence the high percentage of access. The edge labels of the current merged tree would also be used. Due to the manner new internal nodes are assigned indexes as described above, it is not surprising that the current merged tree would have indexes accessed from R_0 more than from R_1, R_1 more than R_2, and so on.

3.4.2 The String Buffer

The access pattern discovered in Fig. 7 indicates that a good buffering strategy for S should

1. Always keep the last partition in the memory, because it is accessed most.
2. As space permits, keep the first several partitions also in the memory, because they are accessed more than subsequent partitions.
3. Have a small page, so we can store the characters fetched of other partitions not located in the memory during the merge.

The first two items are self-explanatory. In our experiment with the Human genome, we allow the first two hundred partitions of size three million characters each to be kept in the memory. The amount of memory required was about 600MB, which is a reasonable amount to assume that a modern computer would have.

When an index of a partition not available in the memory is requested though, the algorithm must fetch the corresponding character from the disk. In fact, we found that a small number of subsequent characters of that index shall soon be accessed after. These characters correspond to an edge string label. Therefore, this substring is fetched and is stored in a page. For simplicity, we keep the page size uniformed. We observed that a page size of 1024 bytes performs well for the human DNA. In order to determine how many pages to use, we conducted several experiments and observed that the pattern of indexes being requested is rather random. Keeping many previous edge labels around in the main memory does not result in fewer misses, therefore we only keep 1 page in the memory.

With the buffer replacement policy implemented, we were able to build a suffix tree of the Human genome.

3.4.3 Storing Prefixed Suffix Trees

Each prefixed suffix tree is written to a binary file in the depth first manner. For an internal node, the starting index, ending index, parent offset, next sibling offset, suffix link ID are stored. The first child always follow its parent node. For a leaf node, everything but the suffix link ID is stored.

As previously stated, when the prefixed suffix sub-trees for each partition are stored, their suffix link information are maintained via node ID. That means, when an internal node v on the final disk-based suffix tree is read, it may have an ID of $sl(v)$. In order to locate $sl(v)$, we need a mechanism that allows ID-to-file offset lookups. Therefore, during the final tree writing to the

disk, we store the internal node ID and file offset pairs onto the disk as well. The pairs are sorted by ID and then written onto binary files, based on the their tree prefixes.

The total number of files for storing a disk-based suffix tree is $2m$, where m is the number of variable-length prefixes. Half of them is for the prefixed suffix trees, and the other half is for the ID-to-file offset pairs.

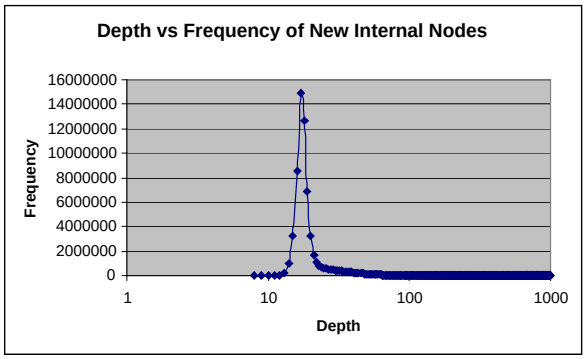

Figure 8: The number of new internal nodes created at different node depths via the merge routine

3.4.4 Time and Space Complexity

The merging algorithm traverses the suffix sub-trees in a depth first manner. Assuming that S is available in the main memory, the algorithm performs constant operations at each step. Since there are $O(n)$ nodes in the suffix trees, the merging phase runs in $O(n)$ time. Note that the time to sort and store the ID and file offset pairs is $O(lg(n))$ and dominated by the $O(n)$ bound.

For the case when S requires the buffer, the run time of this step is $O(n)$ plus the disk I/O time.

The space required during this phase is $O(t)$ plus the amount of memory allocated to the buffer (which is a parameter value that must be set). The $O(t)$ bound emerges from the fact that we create a prefixed suffix sub-tree for each prefix in this phase. Since the variable-length prefixes were created based on the threshold t, therefore the size of each prefixed suffix tree is bounded by $O(t)$.

3.5 Suffix Links

Although the suffix links exist completely in the suffix trees T_{R_i} created for each partition due to the use of Ukkonen's algorithm, they do not equal the set of suffix links if the entire suffix tree were to be created for the full string S. New internal nodes are created during the merge routine and these nodes do not have suffix links. We conducted an experiment to investigate the effect of the missing suffix links using the human chromosome I and t of 3 million. The number of internal nodes created during the merge routine is collected, and displayed in Fig. 8.

The result shows that the nodes at depth approximately 20 are created with the highest frequency, and also the frequency drops sharply as the depth increases. This implies that even though there are a lot of internal nodes without any suffix link information, they are mostly at a short distance from the root.

We further investigate the frequencies of the already existing internal nodes (the ones with suffix links) at different depths. The result shows a very similar pattern to Fig. 8 (its graph is omitted here). This implies that, at the depth where the newly created internal node population is very

dense, the already existing internal node population is very dense also. Therefore, it is likely for a newly created node v to have a parent or a close ancestor with a suffix link. Despite the fact that direct suffix link to *sl(v)* is missing, we might be able to avoid locating *sl(v)* by beginning the search at the root. Instead, we can traverse up from v until we hit an ancestor with a suffix link, travel across that link, and then down to *sl(v)*. Although taking this route is slower than using the suffix link from v directly, it should be faster than searching for *sl(v)* from the root itself.

Note that when locating the *sl(v)* node, we do not search for every characters of the string from the root to *sl(v)*. This is because locating *sl(v)* implies that string from the root to v has already been located in the tree. Since it is a suffix tree, the string from the root to *sl(v)* must certainly exist in the tree. Therefore, the skip and count [36] technique can be applied here. (Skipping and counting means locating the right edge, skip directly to the end of the edge without comparing any edge characters, locating the next right edge, and so on, until the search length is matched.)

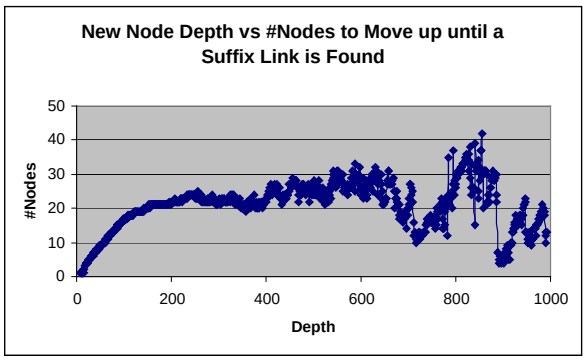

Figure 9: The number of internal nodes to traverse up until we reach a node with a suffix link

To confirm our hypothesis, we collected the number of nodes to traverse up from v until we hit an ancestor of v with a suffix link. Our findings are displayed in Fig. 9. As expected, the number of nodes to traverse up when the depth equals to 20 is minimal, i.e. 4. This is important because 20 is the depth were most new internal nodes are created. The graph shows that the number of nodes to traverse up increases as $d(v)$ increases. However, this comes as a small cost because 1) $d(v)$ is magnitudes higher than the number of nodes to traverse up (and thus to skip and count down) and 2) there are much fewer newly created internal nodes at higher depths.

3.5.1 The Missing Suffix Link Effect

Since the suffix links maintained by Trellis are incomplete, the effect of the missing suffix links on the query time should be studied. Many algorithms, such as MUMmer [12, 28], rely on the suffix links to speed up the search for exact matches between two (or more) strings. The idea is as follows. Let $Q = q_0q_1q_2 \dots q_n$ be a query and suppose $q_0q_1q_2 \dots q_i$, $0 < i \leq n$, is found in the suffix tree. If $i < n$, then q_{i+1} is the first mismatched position. Let the node v be the last internal node accessed before $q_0q_1q_2 \dots q_i$ is matched. Then $q_1q_2 \dots q_i$ can be quickly located in the tree via the suffix link, *sl(v)*, and we then can continue finding the next match from q_{i+1} on. By utilizing the suffix links in this manner, one avoids restarting the search for the next match from the root each time.

Since the suffix links that Trellis has is an incomplete set, we conducted experiments to see if the internal nodes created during the merge phase can rely on the suffix links of their ancestor

nodes instead. We found that the query times via the currently existing suffix links are faster than via no suffix links at all. The experimental results are shown in the next chapter.

4 Experimental Results

4.1 Construction Time

In this section, we present the results of an experimental evaluation of different disk-based suffix tree construction techniques. We compare TRELLIS with DynaCluster and TOP-Q in Figure 10, as they are the only two algorithms that both create suffix links and do not exhibit the data skew problem.

All experiments were performed on an Intel(R) Xeon(TM) processor with 2.80GHz speed, 3GB of main memory, and 32GB of local disk. The implementations of both DynaCluster and TOP-Q were obtained from their respective authors. TRELLIS was written in C++ and compiled with GNU's g++ compiler version 3.2.2 (Red Hat Linux 3.2.2-5) with -O3 optimization flag activated. The input sequence used was the A, C, G, T's of the entire human chromosome I (about 220Mbps).

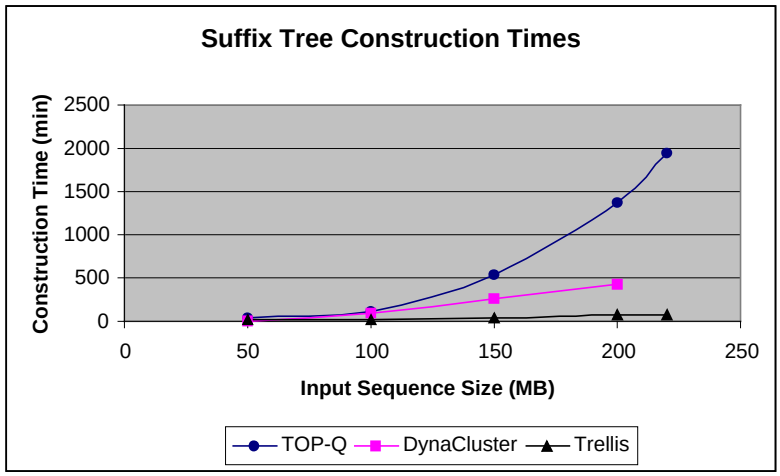

Figure 10: TRELLIS versus DynaCluster and TOP-Q. Total run time to create a disk-based suffix tree with suffix links

In DynaCluster, the LCA method of rebuilding the suffix links was run immediately after the suffix trees were constructed on disk. We experimented with different sizes of LCA buffer, which were 64MB (default), 512MB, 1GB, and 1.5GB, and observed that the latter three gave similar results, which is about $2-6$ times faster than the result of the default size. The buffer size of 512MB gave the best total time, therefore we report its numbers. DynaCluster suffix tree construction phase (without suffix link rebuilt) is insensitive to the amount of memory available, and thus we use the default buffer sizes for the tree construction. The results reported for DynaCluster is the total times to construct the trees and rebuild the suffix links. In general, the suffix tree construction phase of DynaCluster is very fast: the time taken for rebuilding the suffix links was the major part of the total time. DynaCluster did not finish rebuilding the suffix links for the entire chromosome.

In comparison to TOP-Q, we use a buffer pool of 1400MB for internal nodes and 600MB for leaf nodes. (The node sizes in this implementation were 29 and 5 bytes for internal and leaf nodes respectively.) As suggested in [1], the number of internal nodes for typical DNA sequences is $0.6-$

0.8 times the number of leaf nodes, and thus the memory should be allocated accordingly to achieve the best performance possible.

TRELLIS was run with the threshold t of 1 million. The maximum prefix length was six when the input string is the entire chromosome. The memory consumed by TRELLIS was about 100MB during its first phase, and 400MB during the suffix sub-trees merging phrase. TRELLIS was set to keep the entire input string in the memory during the merging phase; about 220MB out of 400MB was used to hold the input string.

As shown in the graph, TRELLIS outperforms both DynaCluster and TOP-Q for the total construction time. TRELLIS performance shows its linear bound of $O(n)$. The reasons why TRELLIS performs better is mainly because it does not need to traverse the entire on-disk tree twice to rebuild the suffix links (as in DynaCluster) and it does not require a tree node buffering scheme because smaller suffix sub-trees are created instead of the entire suffix tree at once (as in TOP-Q).

We also experimented with the entire Human genome. This experiment was conducted on a Power MAC G5 machine with 2 CPUs with a speed of 2.7GHs and 4GB available memory. We chose not to run this under the same cluster machine used for the above experiments because it was a long experiment, which was better run on a dedicated machine. Also more importantly, there was no need to compare this case with either TOP-Q or DynaCluster because neither one of them has reported to finish successfully on the entire Human genome.

We set the threshold t to 3 million. The maximum prefix length was ten. The time taken for generating the variable-length prefixes was about 20 minutes. The time taken for phase 2 and 3 were about 8 and 28 hours respectively. The string buffer was set to hold 200 partitions. The amount of memory consumed by the program was just under 1GB and the size of on-disk suffix tree was about 97GB. Besides TDD [34], we are the only other algorithm reported to finish successfully on the entire Human genome. TDD, however, completely disregards the suffix link structure and therefore serves different purposes from TRELLIS.

Lastly, for the sake of completeness, we also experimented with our implementation of the traditional Ukkonen's algorithm on large input DNA sequences. We found that the program consumes about 1GB of memory already when the input sequence was only 10MB! This confirms the need of a disk-based suffix tree algorithm.

4.2 Missing Suffix Links and Query Time

In this section, we investigate the effect of the missing suffix links on the query time. In this experiment, a set of queries is issued and we compared their search times using the available suffix links vs using no suffix links at all. The idea is that the node v may not have a suffix link, but we may be able to move up for a few nodes before reaching an ancestor node of v such that it has a suffix link and use the link to move across the tree without having to restart the search from the root. Let's denote such ancestor node v_a. Note that if v has a suffix link, then v and v_a are used interchangeably.

For this experiment, we used a disk-based suffix tree for the human chromosome I (about 220MB), with t equals to 3 million. The queries used here are the substrings of the chromosome itself to ensure that they would always be found in the tree. The query length is set to 100, and 500 queries were generated from the substrings starting at the $150,000,000^{th}$ index until the $150,000,499^{th}$ index. The experiment was conducted on the same Power MAC G5 machine. Note that the machine's amount of memory is irrelevant here because the search is done directly from the disk. Note that we conducted the experiments for a variety of query lengths (1K, 10K, 100K, and 1000K) and substring starting positions (50mil, 100mil, 150mil, and 200mil), and results obtained

are similar to the ones shown here.

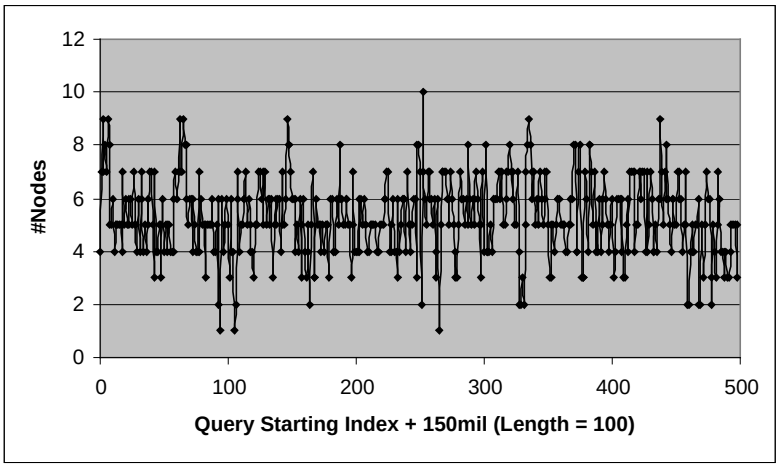

Figure 11: The number of internal nodes to traverse up until we reach a node with a suffix link

The experiment aims at comparing the search time of a given query q by starting the search from the root versus from $sl(v_a)$. To obtain the first set of search times, we simply search starting from the root. As for the second set of timings, we start searching for the first query, q_0, from the root. Once found, we look for the node v_a, obtain $sl(v_a)$'s node ID, then look up its corresponding file offset stored on disk previously, and move across the tree to $sl(v_a)$'s file offset to search for the next query, and so on. The results are shown in Figures 11, 12, and 13.

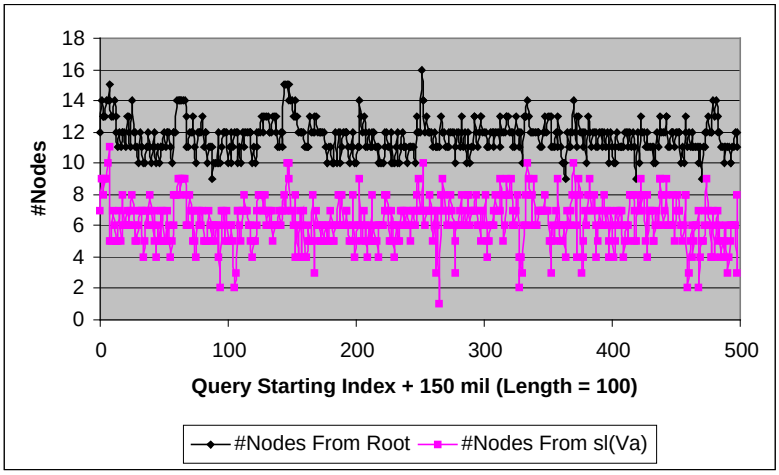

Figure 12: The number of internal nodes to traverse down from the root vs from $sl(v_a)$ until $sl(v)$ is reached

Fig. 11 shows the number of internal nodes to traverse up until we reaches the node v_a. The results conform with our previous discussion such that the number of nodes needed to move up is small.

In Fig. 12, the result shows that, although the link across to $sl(v)$ is not an immediate one, using $sl(v_a)$ still shortens the travel distance considerably.

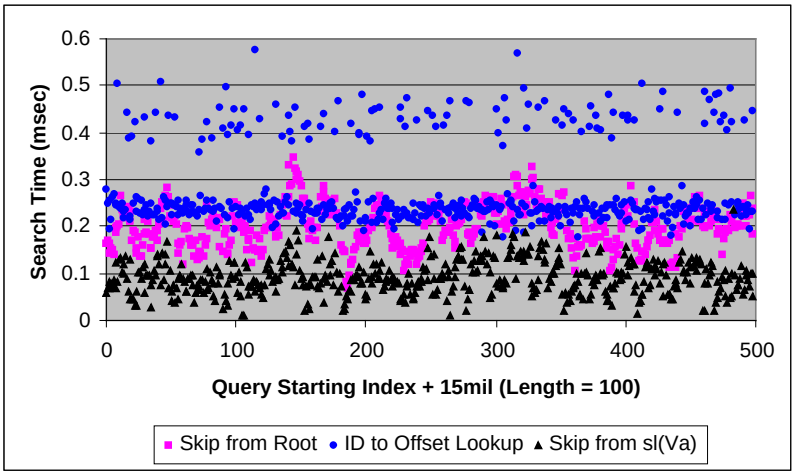

Figure 13: The query time. Although locating *sl(v)* via $sl(v_a)$ is faster than from the root, we first must locate $sl(v_a)$ by looking its offset up via the ID-to-file offset file and this is the bottleneck!

Fig. 13 shows a very interesting behavior regarding the query time. One would expect that, by using $sl(v_a)$ to shorten the travel distance to *sl(v)*, the query time would also be faster. Unfortunately, that is not yet the case because the suffix link ID-to-file offset lookup time dominates the search time. In this experiment, the average times to look up the offset, to find the query by skipping from the root, and to find the query by skipping from $sl(v_a)$ are 0.3, 0.2, and 0.1 msec respectively. The graph shows that, without the offset lookup time, utilizing $sl(v_a)$ results in a faster query time than starting the search from the root. However, that is not the case when we include also the offset lookup time. Therefore, we can conclude from these results that:

- The bottle neck of the query time here is the file offset lookup time, not the skipping-and-counting time from $sl(v_a)$.
- Although the suffix links populated by TRELLIS is incomplete, they still can be utilized to speed up the search time.

Since the suffix link id-to-file offset search time is the bottleneck, an immediate remedy is to perform a post-processing step on the suffix tree that replaces the suffix link IDs with their respective file offsets, so there would be no need to look them up. The post-processing step requires a single suffix tree on-disk traversal. At each internal node, if there is an available suffix link ID, the algorithm fetches its corresponding file offset and replaces the ID with the offset. We added this optional preprocessing step and were able to completely eliminate the suffix ID-to-file offset lookup time added to each query time in Fig. 13. Under the same settings that the trees were constructed above, the additional times taken for the post-processing step were about 10 minutes and 2.5 hours for the Human Chromosome I and the entire Human genome respectively.

In summary, we demonstrated that:

- TRELLIS outperforms TOP-Q and DynaCluster in construction time.
- TRELLIS is currently the only disk-based algorithm that both maintains the suffix links and scales up to the Human genome level.

- Although the suffix links maintained by TRELLIS are partial, they are shown experimentally to enable faster query times than without using any suffix links at all.

5 Conclustion and Future Work

In this work, we developed a novel disk-based suffix tree construction algorithm called TRELLIS. TRELLIS builds the suffix tree based on a partitioning method via variable length prefixes and a suffix sub-trees merging algorithm, which enables it to avoid the data skew problem exhibited in many other disk-based suffix tree methods. In addition, it bypasses the need to use a buffering scheme to manage suffix tree nodes during the tree construction. TRELLIS is a linear time algorithm, given that the input string is stored in the memory, and it does not sacrifice the existence of suffix links in the tree. TRELLIS scales gracefully when the input DNA sequence is large. In fact, we show that a suffix tree on the entire Human genome can be built in about 36 hours, using under 1GB of memory! TRELLIS is currently the only disk-based suffix tree algorithm with suffix links that reports a suffix tree on an input string of this magnitude. Although the suffix links maintained are partial, the query times using them are experimentally shown to be consistently faster than the query times obtained without using any suffix links. TRELLIS is experimentally shown to outperform the other existing algorithms that do not exhibit the data skew problem and also maintain the suffix links in construction time.

As part of our future work, we plan on adapting TRELLIS to a wider range of biological input sequences, e.g. protein sequences. In fact, a previous implementation of TRELLIS was adapted to index successfully the entire SwissProt database [15] using under 1GB of main memory. In this case, each protein character was translated into one of the seven state prediction values generated by HMMSTR [6] (In other words, the alphabet size was seven.) TRELLIS was combined as a backend to the PSIST [20] 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 creating a user interface for TRELLIS. Most of the disk-based suffix tree 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 also be created.

In addition, we plan to build the complete set of suffix links via the partial set maintained currently. As a result, query times should be improved even further.

We also plan to create a buffer to manage the suffix tree nodes during the query process. Although it is important to have an efficient algorithm that constructs a disk-based suffix tree, it is equally critical to be able to *use* the suffix tree from the disk 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 patterns incurred by the algorithm using the disk-based suffix tree.

Lastly, TRELLIS would be compared with other similar large-scale indexing structures, e.g. String B-Trees [19], Suffix Arrays [13], Suffix Vectors [30], and Distributed Suffix Trees [10].

References

[1] Srikanta J. Bedathur and Jayant R. Haritsa. Engineering a fast online persistent suffix tree construction. In *Proceedings of the 20th International Conference on Data Engineering*, pages 720–731, 2004.

[2] Srikanta J. Bedathur and Jayant R. Haritsa. Search-optimized suffix-tree storage for biological applications. In *Proceedings of the 12th IEEE International Conference on High Performance Computing (HiPC)*, pages 29–39, 2005.

[3] Nicolas Bray, Inna Dubchak, and Lior Pachter. AVID: A global alignment program. *Genome Res.*, 13(1):97–102, 2003.

[4] Nicolas Bray and Lior Pachter. MAVID: Constrained ancestral alignment of multiple sequences. *Genome Res.*, 14(4):693–699, 2004.

[5] A. L. Brown. Constructing genome scale suffix trees. In *Proceedings of the 2nd APBC*, pages 105–112, 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] Carvalho, Freitas, Oliveira, and Sagot. Efficient extraction of structured motifs using box-links. In *Proceedings of the 11th Conference on String Processing and Information Retrieval*, volume 11, 2004.

[8] Chang and Lawler. Sublinear approximate string matching and biological applications. *ALGRTHMICA: Algorithmica*, 12:327–344, 1994.

[9] Cheung, Yu, and Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEETKDE: IEEE Transactions on Knowledge and Data Engineering*, 17, 2005.

[10] Raphael Clifford. Distributed suffix trees. *Journal of Discrete Algorithms*, 3(2-4):176–197, June 2005.

[11] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucl. Acids. Res.*, 27(11):2369–2376, 1999.

[12] A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucl. Acids. Res.*, 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] Jitender S. Deogun, Jingyi Yang, and Fangrui Ma. EMAGEN: An efficient approach to multiple whole genome alignment. In *Proceedings of the 2nd APBC*, pages 113–122, 2004.

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

[16] Martin Farach, Paolo Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 174–185, 1998.

[17] Farach-Colton, Ferragina, and Muthukrishnan. On the sorting-complexity of suffix tree construction. *JACM: Journal of the ACM*, 47, 2000.

[18] Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.

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

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

[21] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proceedings of the 3rd Workshop on Algorithm Engineering*, pages 30–42, 1999.

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

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

[24] Michael Höhl, Stefan Kurtz, and Enno Ohlebusch. Efficient multiple genome alignment. In *ISMB*, pages 312–320, 2002.

[25] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. A database index to large biological sequences. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 139–148, 2001.

[26] Robert Japp. The top-compressed suffix tree. In Lachlan M. MacKinnon, Albert G. Burger, and Philip W. Trinder, editors, *Proceedings of the 21st Annual British National Conference On Databases*, volume 2, pages 68–79, Department of Computer Science, School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton Campus, Edinburgh, EH14 4AS, UK, July 2004. Heriot-Watt University.

[27] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw, Pract. Exper*, 29(13):1149–1171, 1999.

[28] Stefan Kurtz, Adam Phillippy, Arthur Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, April 24 2004.

[29] Edward M. McCreight. A space-economical suffix tree construction algorithm. *JACM*, 23(2):262–272, April 1976.

[30] Krisztián Monostori, Arkady B. Zaslavsky, and Heinz W. Schmidt. Suffix vector: Space- and time-efficient alternative to suffix trees. In *Proceedings of the 25th ACSC*, pages 157–165, 2002.

[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] NCBI. Genbank. www.ncbi.nlm.nih.gov/GenBank, 2006.

[33] Klaus-Bernd Schürmann and Jens Stoye. Suffix tree construction and storage with limited main memory. Technical Report 0946-7831, Universität Bielefeld, 2003.

[34] Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical suffix tree construction. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 36–47, 2004.

[35] Yuanyuan Tian, Sandeep Tata, Richard A. Hankins, and Jignesh M. Patel. Practical methods for constructing suffix trees. *VLDB*, pages 281–299, 2005.

[36] Esko Ukkonen. Constructing suffix trees on-line in linear time. In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture: Information Processing*, pages 484–492, 1992.

[37] Esko Ukkonen and Juha Kärkkäinen. Sparse suffix trees. In *Proceedings of the 2nd Annual Intelligence Conference on Computing and Combinatorics*, pages 219–230, 1996.

[38] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.