Genome-scale Disk-based Suffix Tree Indexing

Benjarath Phoophakdee
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, 12180
phoopb@cs.rpi.edu

Mohammed J. Zaki
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, 12180
zaki@cs.rpi.edu

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 paper, we target DNA sequences and propose a novel disk-based suffix tree algorithm called TRELLIS which effectively scales up to genome-scale sequences. Specifically, it can index the entire human genome using 2GB of memory, in about 4 hours and can recover all its suffix links within 2 hours. 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.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Scientific databases*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; J.3 [**Computer Applications**]: Life and Medical Sciences—*Biology and genetics*

General Terms

Algorithms, Experimentation

Keywords

Suffix Tree, Disk-based, External Memory, Genome-scale, Sequence Indexing

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 [25] 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 [17]. Suffix trees have also been extensively used in genome alignment approaches to find matching segments [10, 3, 21].

Suffix trees can be constructed in linear (in the sequence length) time and space [24, 31, 29], provided the tree fits entirely in the main memory. A variety of efficient in-memory suffix tree construction algorithms have been proposed [16, 14, 13, 30, 12]. 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 [14, 22, 28].

Several disk-based suffix tree algorithms have been proposed recently. Some of the approaches [22, 23, 26, 27, 28] 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 [19], structural motifs [5], approximate string matching [6], and genome alignment [10, 3, 21], rely heavily on suffix links for efficiency, and thus cannot be used directly with these disk-based suffix trees. Some approaches [22, 23, 26, 4] 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

[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.

been proposed [1, 7]. However, these methods do not scale up to the human genome level.

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 human genome suffix tree (without the suffix links) [28] 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 [1, 7]. 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.

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.

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 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.

[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

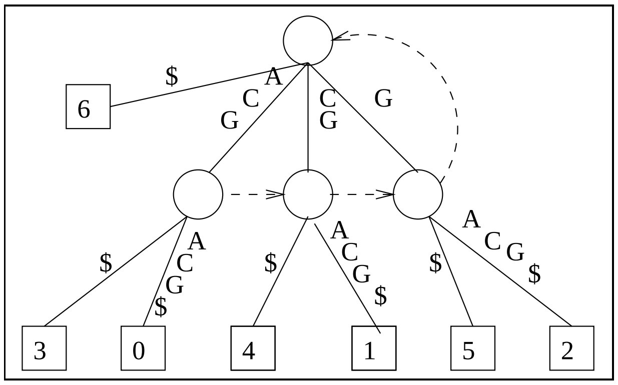

Figure 1: Suffix tree T_S for string S = ACGACG\$. The circles represent internal nodes, and the squares represent leaf nodes. The leaf nodes are numbered with respect to their corresponding suffixes

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 [24, 31, 29], provided the tree fits entirely in the main memory. For example, Ukkonen's [29] 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 in Figure 1. 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 $[3, 5]$. 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. Alternatively, 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 simultaneous use of these three techniques results in linear time and space suffix tree construction.

2.1 Related Work

Hunt et al. [22] introduced one of the first disk-based suffix tree construction algorithms. They 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 the input string, such that the suffix sub-tree from strings having a given prefix, can fit entirely in main memory. 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. Hunt's method has $O(n^2)$ complexity. Hunt's static pre-partitioning method via fixed-length prefixes exhibits 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. 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.

In [26], 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 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.

Top-compressed suffix trees were introduced in [23]. 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.

Paged and distributed suffix trees were proposed in [8]. 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).

Cheung et al. [7] addressed the skew problem via dynamic clustering. In DynaCluster [7], the suffix tree is created one cluster at a time. The clusters provide 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; thus its complexity is $O(n^2)$. An optional phase that rebuilds the suffix links after the entire tree is constructed on disk is provided. However, its suffix link rebuild time is reported to be much slower than its tree construction time.

Bedathur et al. [1] developed the TOP-Q buffer management policy for online disk-based suffix tree construction. TOP-Q is built upon the linear time Ukkonen [29] algorithm and thus results in a disk-based suffix tree construction algorithm that does not sacrifice the suffix links. 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. 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 [27, 28]. 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 [16], combined with a specialized buffering strategy that allows better cache usage during tree construction. TDD was experimentally shown to outperform both Hunt's approach and TOP-Q. It is currently 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 [28]. 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. While ST-Merge was experimentally shown to outperform TDD when the input string is much larger than the main memory, both TDD and ST-Merge suffer the same drawback, which is the lack of suffix links.

Note that other suffix trees variants [20], and other disk-based sequence indexing structures like String B-trees [15] and external suffix arrays [11, 9] have also been proposed to handle large sequences.

3. 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 2.

3.1 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

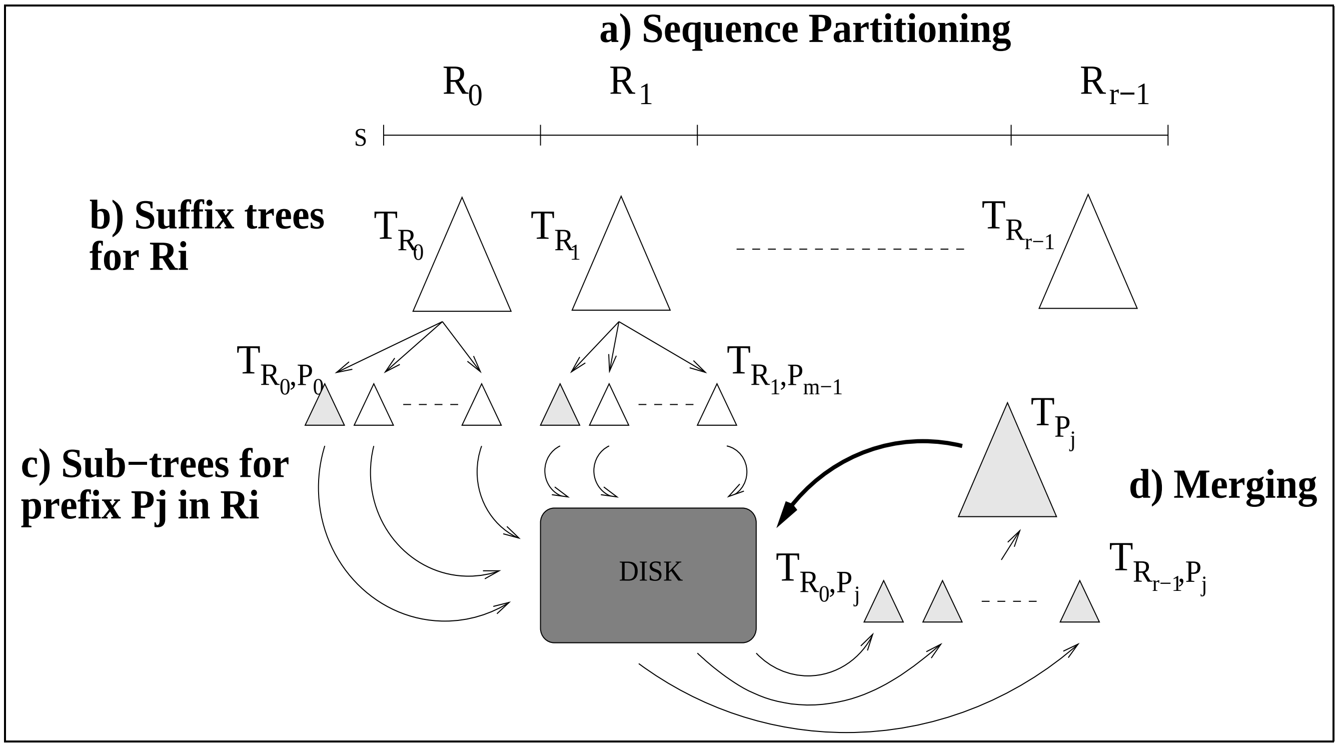

Figure 2: Overview of TRELLIS: Given input sequence S, we first partition it into segments R_i (step a). The segment size is chosen (based on a threshold t) such that the resulting suffix tree T_{R_i} from each segment (step b) fits in main memory. Each resulting suffix tree is further split into smaller 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}).

originally introduced in [22]. 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 [7]. 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 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 [22], 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 $P = \{P_0, P_1, P_2, \dots, P_{m-1}\}$ 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 3.5). Prefix frequency statistics from the human genome are shown in Table 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 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 1, only two prefixes of length 8 have frequency $> t$, and at length 16, none remain (since $4^{16} > 3$Gbp). 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.

3.2 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 [29] 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 2 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.

3.3 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.

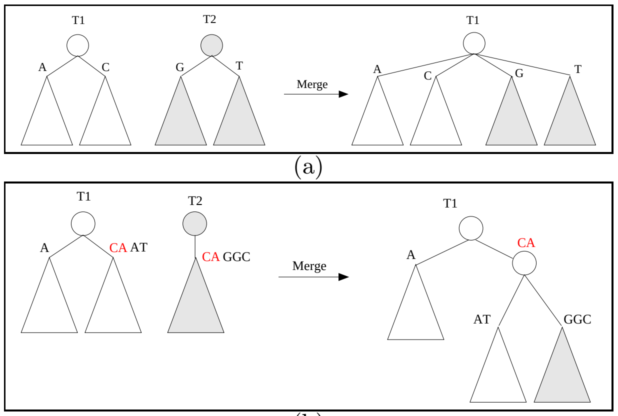

Figure 3: Merging Trees: (a) simple case, with disjoint labels. (b) case when some edges have a common prefix.

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 3 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 3(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 $(3 \times 10^9)/4$ 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 [28]. 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.

3.4 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 [3, 10, 21] and tandem repeats search [19]. 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 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, $sl(p(v))$. This suffix link may point to another prefix-subtree T_{P_a}, which is loaded into memory the first time it is accessed. Next the algorithm skip/counts [29] down from $sl(p(v))$ to locate $sl(v)$. Once found, the disk location (physical pointer) of $sl(v)$ is added to v, and the algorithm then proceeds recursively to all children of v that are internal nodes. 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 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}, $\ldots$, 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 3.5), this step can operate under the given amount of available memory.

3.5 Choosing the 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:

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

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.

3.6 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 [29]. 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 3.2, 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 [17, theorem 6.1.1], that over the entire suffix tree the skip/count steps takes at most $3n = O(n)$ time. 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 $O(n)$.

4. EXPERIMENTAL RESULTS

In this section, 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 [1] and DynaCluster [7]. Then, we compared it with the best known algorithms that do *not* maintain suffix links, namely TDD [27, 28] and ST-Merge [28]. Note that TDD and ST-Merge have been shown to outperform the former two algorithms by substantial margins.

All of the experiments 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.

4.1 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 2: 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 2 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 3), 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 2 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.

4.2 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 3. 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 3 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

Table 3: Suffix tree construction times for Trellis and TDD

Input Length (Mbp)	Memory Available (MB)	**TDD** Construction (mins)	Trellis Construction (mins)	Trellis Link Recovery (mins)	Trellis 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	12.8 hours	4.2 hours	1.7 hours	5.9 hours

Table 4: 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
Human Genome	71.6	27.1	54.0	19.3

Table 5: Memory-based Suffix Tree Construction

Input	Running Time		Memory Usage	
String (Mbp)	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

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 4) 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 4 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 [16]. 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 [16] method, and Trellis becomes equivalent to the Ukkonen's [29] 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 5 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 2 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, respectively (see Table 4). 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 [28] 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 [28], 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 3, 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 [17]. 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.

4.3 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 [10]. 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.

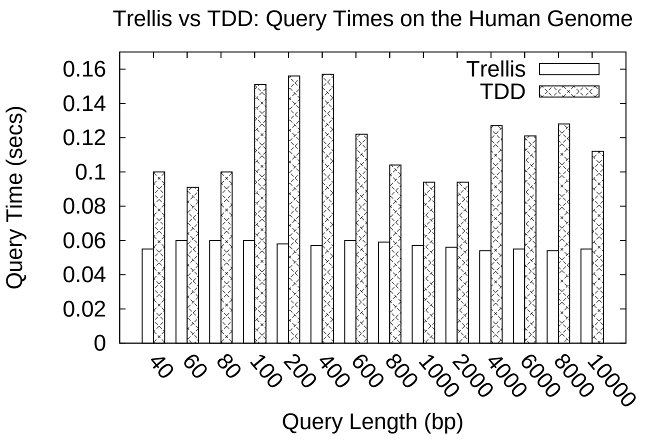

Figure 4: Average times of 500 random queries on the human genome suffix tree.

Figure 4 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 4). 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.

4.4 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 align-

[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.

[4]Personal communication.

ment program (see [10] for details). To briefly describe the algorithm, let $Q = q_0q_1 \dots q_{|Q|-1}$ 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.

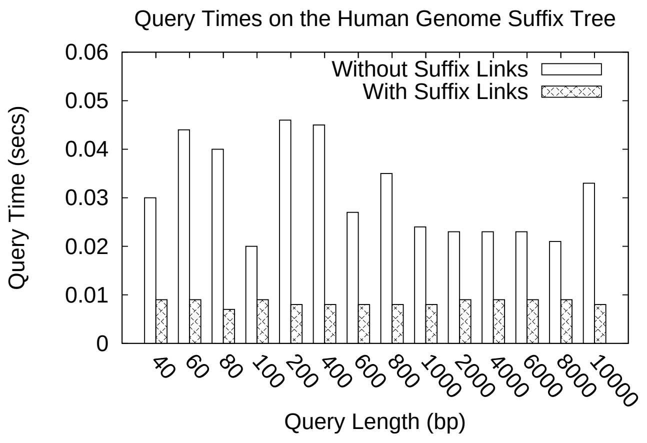

Figure 5: Average time on 500 consecutive queries, with and without suffix links, on the human genome suffix tree.

We ran query experiments on the entire human genome suffix tree (see Figure 5). 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 into the human genome sequence, for a query of length l, we issued 500 queries, Q_j (with $j \in [0, 500)$), where $Q_j = s_{i+j}s_{i+j+1} \dots s_{i+j+l-1}$. 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 5 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 4, TDD average query times are between 0.1-0.15s, whereas Trellis with suffix links takes on average 0.01s.

5. CONCLUSIONS AND FUTURE WORK

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 [25]. 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 [18]. 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 [28]. 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.

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. Additionally, we plan to develop a buffering strategy for the input sequence. Currently, the human genome is the largest publicly available genome. With bit-encoding of the DNA alphabet, approximately 750MB of main memory is required to entirely store the genome during the tree construction. Since TRELLIS targets DNA sequences and the above amount of memory is reasonable to assume on modern computers, there is no need for a string buffering strategy to achieve our present goals. However, if we want to build a generalized suffix tree composed of several large genomes, the combined string length would exceed the main memory and thus require some buffering strategy. Finally, we plan to parallelize TRELLIS which follows as a logical next step, since its partitioning and merging steps seem ideally suited to parallelization.

Acknowledgments

We would like to thank the authors of TOP-Q [1], DynaCluster [7] and TDD [27, 28] for giving us their codes. We also like to thank the anonymous reviewers for their helpful suggestions.

6. REFERENCES

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

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

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

[5] 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.

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

[7] C.-F. Cheung, J. 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.

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

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

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

[11] R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *Workshop on Algorithm Engineering and Experiments*, 2005.

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

[13] 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.

[14] 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.

[15] P. Ferragina and R. 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.

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

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

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

[19] 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.

[20] 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.

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

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

[23] 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.

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

[25] 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.

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

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

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

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

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

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