TRELLIS+: AN EFFECTIVE APPROACH FOR INDEXING GENOME-SCALE SEQUENCES USING SUFFIX TREES *

BENJARATH PHOOPHAKDEE AND MOHAMMED J. ZAKI
Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 12180
E-mail: {phoopb,zaki}@cs.rpi.edu

With advances in high-throughput sequencing methods, and the corresponding exponential growth in sequence data, it has become critical to develop scalable data management techniques for sequence storage, retrieval and analysis. In this paper we present a novel disk-based suffix tree approach, called TRELLIS+, that effectively scales to massive amount of sequence data using only a limited amount of main-memory, based on a novel string buffering strategy. We show experimentally that TRELLIS+ outperforms existing suffix tree approaches; it is able to index genome-scale sequences (e.g., the entire Human genome), and it also allows rapid query processing over the disk-based index. **Availability:** TRELLIS+ source code is available online at `http://www.cs.rpi.edu/~zaki/software/trellis`

1. Introduction

Sequence data banks have been collecting and disseminating an exponentially increasing amount of sequence data. For example, the most recent release of GenBank contains over 77 Gbp (giga, i.e., 10^9, base-pairs) from over 73 million sequence entries. Anticipated advances in rapid sequencing technology, applied to metagenomics (i.e., study of genomes recovered from environmental samples) or rapid, low-cost human genome sequencing, will yield a vast amount of short sequence reads. Individual genomes can also be enormous (e.g., the *Amoeba dubia* genome is estimated to be 670 Gbp [a]). It is thus crucial to develop scalable data management techniques for storage, retrieval and analysis of complete and partial genomes.

In this paper we focus on disk-based suffix trees as the index structure for effective massive sequence data management. Suffix trees have been used to efficiently solve a variety of problems in biological sequence analysis, such as exact and approximate sequence matching, repeat finding, and sequence assembly (via all pairs suffix-prefix matching) [9], as well as anchor finding for genome alignment [4]. Suffix trees can be constructed in time and space linear in the sequence length [16], provided the tree fits entirely in the main memory. A variety of efficient in-memory suffix tree construction algorithms have been proposed [8,6]. However, these algorithms do not scale up when the input sequence is extremely large.

Several disk-based suffix tree algorithms have been proposed recently. Some of the approaches [11,12,15] completely abandon the use of suffix links

*This work was supported in part by NSF Career award IIS-0092978, and NSF grants EIA-0103708 and EMT-0432098.
[a]Database of Genome Sizes: `http://www.cbs.dtu.dk/databases/DOGS/`

and sacrifice the theoretically superior linear construction time in exchange for a quadratic time algorithm with better locality of reference. Some approaches [11,12,2] also suffer from the skewed partitions problem. They build prefix-based partitions of the suffix tree relying on a uniform distribution of prefixes, which is generally not true for sequences in nature. This results in partitions of non-uniform size, where some are very small, and others are too large to fit in memory. Methods that do not have the skew problem and that also maintain suffix links, have also been proposed [1,3]. However, these methods do not scale up to the human genome level. The only known suffix tree methods that can handle the entire human genome include TDD [15] and TRELLIS [13]. TRELLIS was shown to outperform TDD by over 3 times. However, these methods still assume that the input sequence can fit in memory, which limits their suitability for indexing massive sequence data. Other suffix trees variants [10], and other disk-based sequence indexing structures like String B-trees [7] and external suffix arrays [5,14] have also been proposed to handle large sequences. A comparison between TDD [15] and the DC3 [5] method for disk-based suffix arrays suggests that TDD is twice as fast [15].

In this paper we present a novel disk-based suffix tree indexing algorithm, called TRELLIS+, for massive sequence data. TRELLIS+ effectively handles genome-scale sequences and beyond with only a limited amount of main-memory. We show that TRELLIS+ is over twice as fast as TRELLIS, especially with restricted amount of memory. TRELLIS+ is able to index the entire human genome (approx. 3Gbp) in about 11 hours, using only 512MB of memory, and on average queries take under 0.06 seconds, over various query lengths. To the best of our knowledge these are the fastest reported time with such a limited amount of main-memory.

2. Preliminary Concepts

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 $S = s_0 s_1 s_2 \dots s_{n-1}$ be the input string where $S \in \Sigma^\star$ and its length $|S| = n$. The i^{th} suffix of S is represented as $S_i = s_i s_{i+1} s_{i+2} \dots s_{n-1}$. 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 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,

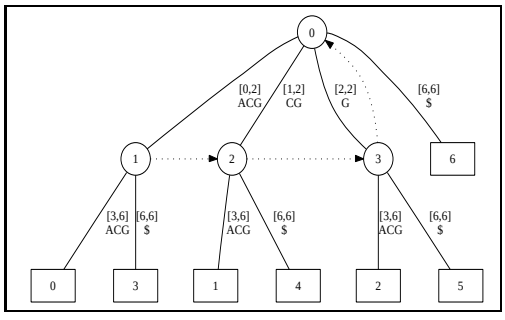


Figure 1. Suffix tree T_S for S = ACGACG$.

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.

Let $\sigma(v)$ denote the substring obtained by concatenating all characters from the root to node v. Each internal node v also maintains a *suffix link* to the internal node w, where $\sigma(w)$ is the immediate suffix of $\sigma(v)$. A suffix tree example is given in Fig. 1; circles represent internal nodes, square nodes denote leaves, and dashed lines indicate suffix links. Internal nodes are labeled in depth-first order, and leaf nodes are labeled by the suffix start position. The edges are also shown in the encoded form, giving the start and end positions of the edge label.

3. The Basic Trellis+ Approach

TRELLIS+ follows the same overall approach as TRELLIS [13]. Let S denote the input sequence, which may be a single genome, or the string obtained by concatenating many sequences. TRELLIS+ follows a partitioning and merging approach to build a disk-based suffix tree. The main idea is to maintain a complete suffix tree as a collection of several prefix-based sub-trees. TRELLIS+ has three main steps: i) prefix creation, ii) partitioning, and iii) merging.

In the *prefix creation phase* TRELLIS+ creates a list of variable-length prefixes $\{P_0, P_1, \cdots, P_{m-1}\}$. Each prefix P_i is chosen so that its frequency in the input string S does not exceed a maximum frequency threshold, t_m, determined by the main-memory limit, which guarantees that the prefix-based sub-tree T_{P_i}, composed of all the suffixes beginning with P_i as a prefix, will fit in the available main-memory. The variable prefix set is computed iteratively; in each iteration prefixes up to a given length are counted (those that exceed the frequency threshold t_m in the last iteration).

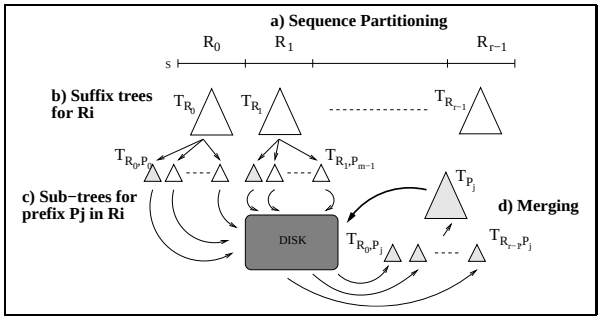


Figure 2. Overview of TRELLIS+

In the *partitioning phase*, the input string S is split into $r = \lceil \frac{n+1}{t_p} \rceil$ segments (Fig. 2, step a), where $n = |S|$ and t_p is the segment size threshold, chosen so that the resulting suffix tree T_{R_i} for each segment R_i (Fig. 2, step b) fits in main-memory. Note that T_{R_i} contains all the suffixes of S that start only in segment R_i; T_{R_i} is constructed using the in-memory Ukkonen's algorithm [16]. Each resulting suffix tree T_{R_i} from a given segment is further split into smaller subtrees T_{R_i,P_j} (Fig. 2, step c), that share a common

prefix P_j, which are then stored on the disk.

After processing all segments R_i, in the *merging phase*, TRELLIS+ merges 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} (Fig. 2, step d). Note that T_{P_j} is guaranteed to fit in memory due to the choice of t_m threshold. 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} for segments R_0 through R_i. In the next step we merge T_{R_{i+1},P_j} from segment R_{i+1} with M_i to obtain M_{i+1}, and so on (for $i \in [0, r-1]$). The merging is done recursively in a depth-first manner, by merging labels on all child edges, from the root to the leaves. 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 complete suffix tree is simply a forest of these prefix-based subtrees (T_{P_j}). Note that TRELLIS+ has an optional *suffix link recovery phase*, but we omit its description due to space limitations; see [13] for additional details.

4. Trellis+: Optimizations for Massive Sequences

In this section, we introduce two optimizations to the original TRELLIS. The first optimization is based on a simple observation that *larger* suffix subtrees can be created in the partitioning phase under the *same* memory restriction. As a result, there is less disk management overhead, and fewer merge operations are required, speeding up the algorithm. The second optimization is a novel string buffering strategy. The buffer is based on several techniques, which together remove the limitation of TRELLIS that requires the input sequence to fit entirely in memory. This means TRELLIS+ can index sequences that are much larger than the available memory.

4.1. *Larger Segment Size*

TRELLIS+ uses two thresholds, t_p and t_m, to ensure that the suffix subtrees for a given segment T_{R_i} and a given prefix T_{P_j}, respectively, can fit in memory. Let $|S| = n$ be the sequence length, M be the available main-memory (in bytes), and let s_i and s_l be the size of an internal and leaf node. Typically, the number of internal nodes in the suffix tree is about 0.8 times the number of leaf nodes. During the partitioning phase, the sequence corresponding to the segment R_i is kept in memory in a compressed form, costing $t_p/4$ bytes space (since we use 2 bits to encode each of the 4 DNA bases). Since T_{R_i} has t_p leaf nodes and $0.8t_p$ internal nodes, t_p is chosen to satisfy the following equation:

$$M \geq \frac{t_p}{4} + (0.8s_i + s_l)t_p \implies t_p \leq \frac{M}{\frac{1}{4} + (0.8s_i + s_l)} \tag{1}$$

During the merging phase, we use the threshold t_m to ensure that T_{P_j} can fit in memory. T_{p_j} has t_m leaf and $0.8t_m$ internal nodes. Additionally, new internal nodes, on the order of $0.6t_m$, are created during the edge merge

operations. Furthermore, since all segments can be accessed, we would need to keep the entire input string S in memory, taking up space $n/4$ bytes (this limitation will be removed in Sec. 4.2). Thus t_m is chosen to satisfy the following equation:

$$M \geq \frac{n}{4} + (0.8s_i + s_l + 0.6s_i)t_m \implies t_m \leq \frac{M - \frac{n}{4}}{(1.4s_i + s_l)} \tag{2}$$

TRELLIS uses a global threshold $t = \min(t_p, t_m)$ to control the overall memory usage. However, note that t_m is always smaller than t_p (since $t \ll n$), and this means that as the input sequence length increases, TRELLIS must choose smaller and smaller thresholds, resulting in a corresponding increase in the number of segments, degrading the overall index construction time.

Our first optimization is based on a simple but effective observation that the partitioning phase need not use the global t threshold. TRELLIS+ uses the larger t_p value for the partitioning phase, since Eq. (1) already guarantees that T_{R_i} will fit in M bytes. For the merging phase TRELLIS+ uses the smaller t_m value given by Eq. (2) to guarantee that each T_{P_j} fits under M. This means that TRELLIS+ uses fewer, larger partitions, resulting in fewer tree merge operations, and fewer disk I/O operations, yielding faster overall running times. Note however that there is no difference in the number of variable length prefixes, since the same threshold $t = t_m$ is used.

4.2. *The String Buffer*

During the partitioning phase TRELLIS+ needs to keep the current input string segment R_i in memory. However, for the merging phase, without any optimization, TRELLIS+ would require the entire input string in memory. To remove this memory bottleneck, TRELLIS+ uses a novel string buffering technique, which requires only a small amount of memory to be assigned to the input string during the merging phase, thus enabling TRELLIS+ to scale to extremely large sequences. The string buffering strategy relies on several different techniques, each uniquely important because of its impact on the buffer hit rate. The basic idea behind the buffer design is to keep the characters most likely to be accessed in memory, and to load the rest from disk as needed.

4.2.1. *Edge Index Shifting*

The goal of the index shifting technique is to restrict the character accesses during the merging phase to a small region of the input sequence. This small region of the input string can then be kept in memory as a part of the string buffer, hence increasing the buffer hit rate. Recall that a suffix tree edge is represented by two indexes, $[start, end]$, denoting its edge label $S[start \ldots end]$. The basic observation is that these indexes need not be unique so long as they denote the same string label.

For example, an edge with label "AT" may use the indexes $[0, 1]$ or $[1000, 1001]$ to encode its label, as long as $S[0] = A, S[1] = T$, and $S[1000] = A$, $S[1001] = T$. Another important observation is that the edge lengths between two internal nodes, i.e., internal edge lengths, are generally short. For example, using Human Chromosome I (approx. 200Mbp), we found that most internal edge lengths fall between 1 and 25 characters, and the majority are only a few characters long (the mean length is only 6.7), as shown in Fig. 3.

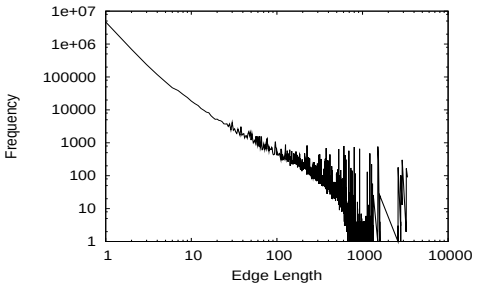
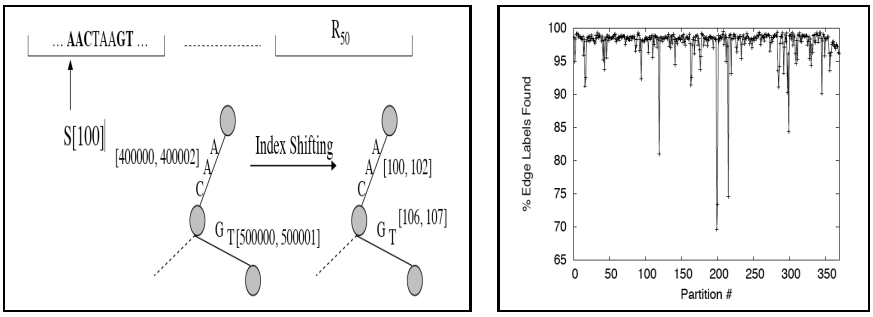


Figure 3. Distribution of internal edge lengths



(a) (b)

Figure 4. (a) Index Shifting, (b) Percentage of Indexes Shifted

To implement the index shifting technique, a small "guide" suffix tree is independently maintained, built from the first 2Mbp of Human Chromosome I. Prior to writing each internal edge in any subtree T_{R_i} to the disk, we search for its string label in the guide suffix tree. If found, we switch the edge's current indexes to the indexes found in the guide tree. The edge index shifting is illustrated in Fig. 4(a); here, two edges from the partition R_{50} have their edge indexes shifted to indexes at the beginning of the input string.

Based on the data from all the partitions for the complete Human genome (using 512MB memory), as shown in Fig. 4(b), we found that on average 97% of the internal edge label indexes can be shifted to the range $[0 \ldots 2 \times 10^6)$ via this optimization. This behavior is not entirely surprising, since the genome contains many short repeats, most of which are likely to have been encountered in the first 2Mbp segment of the genome (which is confirmed by Fig. 4(b)). In addition to the guide tree, the string $S[0 \ldots 2 \times 10^6)$ is also stored in the memory (requiring 0.5MB space after compression) as part of the string buffer because it will be heavily accessed during the merging step. The guide suffix tree requires about 70MB mem-

ory. Furthermore, as mentioned previously, additional internal nodes are created during the subtree merging phase. TRELLIS+ also shifts these indexes to be in the range $[0 \ldots 2 \times 10^6)$.

4.2.2. *Buffering Internal Edge Labels*

Fig. 4(b) shows that approximately 3% of the internal edge labels are still not found in the guide suffix tree. These leftover pairs of internal edge indexes are recorded during the partitioning phase whenever index shifting cannot be applied. Then, during the merging phase, the substrings corresponding to these index ranges are loaded directly into the main memory. These strings are also compressed using 2 bits per character. In all of our experiments (even for the complete human genome), the memory required to keep these substrings consumes at most 20MB.

4.2.3. *Buffering Current Segment*

Subtrees T_{R_i,P_j} are always merged starting from segment R_0 to the last partition R_{r-1} for each prefix P_j. When the i^{th} subtree is being merged with the intermediate merged prefix-subtree M_{i-1} (from partitions R_0 through R_{i-1}), the substring from partition R_i is more heavily accessed than those of the previous partitions. Based on this observation, TRELLIS+ always keeps the string corresponding to the current partition R_i in memory, which requires $\frac{t_p}{4}$ bytes of space.

4.2.4. *Leaf Edge Label Encoding*

The index shifting optimization can only be applied to internal nodes, and not to the leaf nodes, since the leaf edge lengths are typically an order of magnitude longer than internal node edge lengths. Nevertheless, we observed that generally only a few characters from the beginning of the leaf edges are accessed during merging (before a mismatch occurs). This is because leaves are relatively deep in the tree and lengthy exact matches do not occur too frequently. Therefore, merging does not require too many leaf character accesses. To guarantee that the more frequently accessed characters are readily in memory, we allow 64 bits to store the first 29 characters (which require 58 bits, with 2 bits per character) of each leaf label. The last 6 bits are used as an offset to denote the number of current valid characters for the leaf edge. Initially all 29 characters are valid, but characters towards the end become invalid if an internal node is created as a result of merging the leaf edge with another edge. The encoded strings are stored with their respective leaf nodes, and not actually in the memory buffer. Since disk accesses are expensive, the encoded strings are loaded on an as needed basis (we found that 15 – 35% of leaves are not accessed at all during the merge). The memory required for leaf edge label encoding is at most $8t_m$ bytes per prefix. We found that about 93 – 97% of leaf characters accessed during the merge can be found using the encoded labels.

4.2.5. *String Buffer Summary*

As for the rest of the characters that are a buffer miss (i.e., not captured by any of the above optimizations), they are directly read from the disk. We found that the input sequence disk access pattern resulting from the buffer misses during the merge has very poor locality of reference, i.e., it is almost completely random, with the exception that short consecutive range of characters are accessed together. These short ranges represent the labels of the edges being merged. Therefore, we keep a small label buffer of size 256KB to store the characters that require a direct disk access: each disk read fetches 256KB consecutive characters at a time.

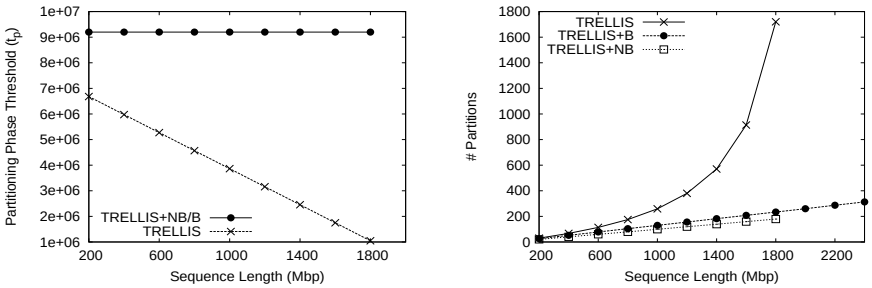
The total amount of memory required for all of the optimization constituting the string buffer can be calculated by adding the amounts of memory required for each technique: 0.5MB for the index shifting, 70MB for the guide tree, 20MB for buffering internal edge labels, $\frac{t_p}{4\times 10^6}$MB for buffering current segment, $\frac{8t_m}{10^6}$MB for leaf edge label encoding, and 0.25MB for the small label buffer. The total string buffer size is thus well under 100MB, using 512MB memory limit (using Eqs.(1) and (2) to compute t_p and t_m). Note that like TRELLIS, TRELLIS+ has $O(n)$ space and $O(n^2)$ time complexity in the worst case, due to the $O(n^2)$ worst-case merging phase time. In practice the running time is $O(n \log n)$; see [13] for a detailed complexity analysis of TRELLIS.

5. Experiments

We now present an experimental study on the performance of TRELLIS+. We compare TRELLIS+ only against TRELLIS since we showed [13] that TRELLIS outperforms other disk-based suffix methods like TDD [15], DynaCluster [3], TOPQ [1] and so on. TDD [15] was in turn shown to have much better performance than the Hunt's method [11], and even a state-of-the-art suffix array method, DC3 [5]. Note that we were not able to compare with ST-Merge [15] (an extension of TDD, designed to scale to sequences larger than memory), since its implementation is not currently available from its authors. All experiments were performed on an Apple Power Mac G5 machine with 2.7GHz processor, 512KB cache, 4GB main-memory, and 400GB disk space. The maximum amount of main-memory usage across all experiments was restricted to 512MB; this memory limit applies to all internal data structures including those for the suffix tree, memory buffers and the input string. Both TRELLIS+ and TRELLIS were compiled with the GNU g++ compiler v. 3.4.3 and were run in 32-bit mode; they produce identical suffix trees. The sequence data used in all experiments are segments of the human genome ranging from size 200Mbp to 2400Mbp, as well as the entire human genome. To study the effects of the two optimizations, we denote by TRELLIS+NB the version of TRELLIS+ that only has the large segment size optimization but no string buffer, and we denote by TRELLIS+B, the version that has both the larger segment and string buffer optimizations.

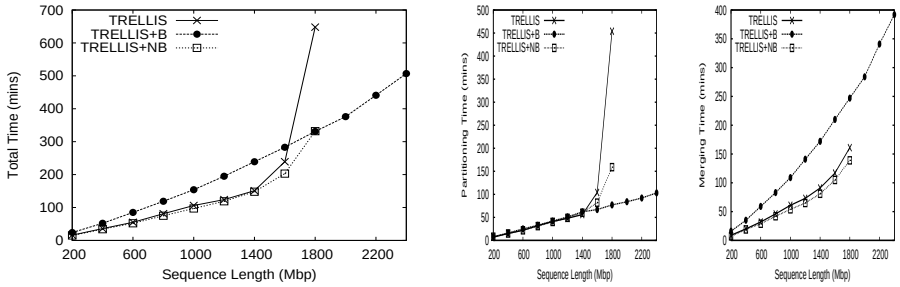
5.1. *Effect of Larger Segment Size*

Here we study the effects of the larger segment size, without the string buffer. TRELLIS+NB has larger and therefore fewer partitions than TRELLIS, since for TRELLIS the number of partitions is $O(\frac{n}{t_p})$ and the value of t_p decreases as the sequence length n increases, resulting in many partitions (as shown in Fig. 5(a)). Therefore, when indexing a very large sequence, the performance of TRELLIS suffers when t_p is small, because of a large number of partitions. In contrast, since the partitioning threshold t_p for TRELLIS+NB remains constant regardless of n, its number of partitions increases at a much slower rate, as shown in Fig. 5(b).



(a) Partitioning Threshold (t_p) (b) Number of Partitions

Figure 5. Effect of Larger Segment Size on Partitioning Phase

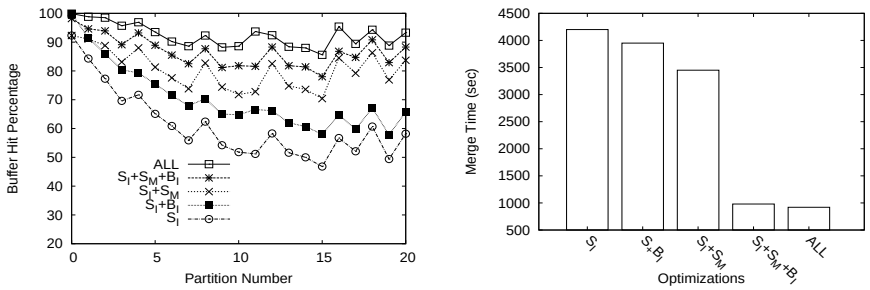


(a) Total Running Time (mins) (b) Partitioning Time (c) Merging Time

Figure 6. Running Time Comparison

The timings of TRELLIS+NB in comparison to TRELLIS are shown in Figs. 6(a), 6(b), and 6(c), which show the total time, partitioning phase time, and merging phase time for TRELLIS+NB versus TRELLIS, as we increase the sequence length from 200Mbp to 1.8Gbp. We find that TRELLIS+NB consistently outperforms TRELLIS, especially when the input sequence size is much larger than the available memory (which is only 512MB). For example, TRELLIS+NB is about twice as fast as TRELLIS for the 1.8Gbp input sequence. This is directly a consequence of the larger,

fewer partitions used by TRELLIS+NB, which result in a much faster partitioning phase (see Fig. 6(b)). The impact of larger segment sizes on the merging phase is not much (see Fig. 6(c)), but TRELLIS+NB still has faster merge times, since there are fewer partitions to be merged for each prefix-based subtree T_{P_j}.



(a) Buffer Hit Rate (b) Buffer Optimizations Times

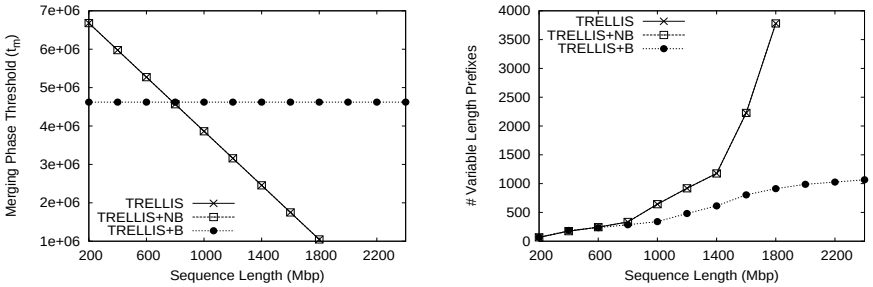
Figure 7. Effect of String Buffer Optimizations

5.2. *Effect of String Buffer*

We now investigate the effect of the string buffering strategy. First we report the difference in the buffer hit rate and merging phase time for TRELLIS+B using the different combinations of buffering optimizations. Fig. 7(a) shows the buffer hit rate for all the characters accessed during the subtree merging operations, using as input string Human Chromosome I (with length approx. 200Mbp), with the 512MB memory limit. The hit rates are shown only for the first 20 partitions, but the same trend continues for the remaining partitions. In the figure, S_I denotes the internal edge index shifting, S_M denotes index shifting during merge phase, B_I denotes buffering internal labels, and ALL denotes all the buffering optimizations. We can clearly see that internal edge index shifting alone yields a buffer hit rate of over 50%. Combination of optimizations yield higher hit rates, so that when all the optimization are combined we achieve a buffer hit rate of over 90%. Fig. 7(b) shows effect of the improved buffer hit rates on the running time of the merging phase in TRELLIS+B. All the optimizations results in a four-fold decrease in time.

Comparing the total running time, and the times for the partitioning and merging phases (shown in Figs. 6(a), 6(b), and 6(c)), we find that initially TRELLIS+NB (that does not use the string buffer) outperforms TRELLIS+B (that uses string buffer). However, as the input sequence becomes much larger, TRELLIS+NB is left with less memory to construct the tree, because it has to maintain the entire compressed input string in memory. Consequently, beyond a certain sequence length, TRELLIS+B starts to outperform TRELLIS+NB. In fact, without string buffer, we were not able to run TRELLIS+NB on an input of size larger than 1.8Gbp, whereas with the string buffer TRELLIS+B can construct the disk-based suffix tree for

the entire Human genome. For a 2.4Gbp sequence, TRELLIS+B took about 8.3 hrs (500 mins, as shown in Fig. 6(a)), and *for the full Human genome (with over 3Gbp length),* TRELLIS+B *finished in about 11 hours using only 512MB memory!* [b]



(a) Merging Threshold (t_m) (b) Number of Prefixes

Figure 8. Effect on the Merging Threshold and Number of Variable Length Prefixes

Fig. 8(a) shows the merging phase threshold t_m, and Fig. 8(b) shows the number of variable-length prefixes for TRELLIS+B and TRELLIS+NB. Since TRELLIS+NB has to retain the entire input string in memory during the merging phase, with increasing sequence length TRELLIS+NB has less amount of memory remaining, resulting in smaller t_m and many more prefixes. On the other hand, for TRELLIS+B the number of prefixes grows very slowly. Overall, as shown in Figs. 6(b) and 6(c), the suffix buffer allows TRELLIS+B to scale gracefully for sequence much larger than the available memory, whereas TRELLIS+NB could not run for an input string longer than 1.8Gbp (with 512MB memory).

5.3. *Query Times*

We now briefly discuss the query time performance on the disk-based suffix tree created by TRELLIS+ on the entire human genome (which occupies about 71GB on disk). 500 queries of different lengths ranging from 40bp to 10,000bp were generated from random starting positions in the human genome. Figure 9 shows the average query times over the 500 random queries for each query length (using 2GB memory). *The average query time for even the longest query (with length 10,000bp) took under 0.06s, showing the effectiveness of disk-based suffix tree indexing in terms of the query performance* (see [13] for more details).

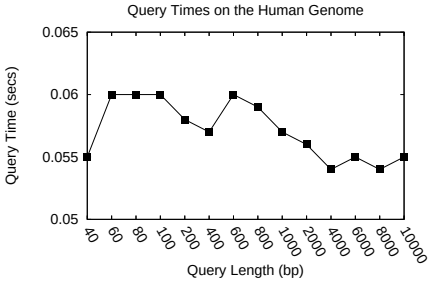


Figure 9. Average Query Times

[b] We showed earlier [13] that TRELLIS can index the entire human genome in about 4 hours with 2GB memory.

6. Conclusion

In this paper we have presented effective optimization strategies which enable TRELLIS+ to handle genome-scale sequences, using only a limited amount of main memory. TRELLIS+ is suitable for indexing entire genomes, or massive amounts of short sequence read data, such as those resulting from cheap genome sequencing and metagenomics projects. For the latter case, we simply concatenate all the short reads into a single long sequence S and index it. In addition we maintain an auxiliary index on disk that allows one to look up for each suffix position S_i, the corresponding sequence id, and offset into the short read. Using all pairs suffix-prefix matching [9], our disk based suffix tree index can enable rapid sequence assembly, and can also enable other next generation sequence analysis applications.

References

1. S.J. Bedathur and J.R. Haritsa. Engineering a fast online persistent suffix tree construction. In *20th Int'l Conference on Data Engineering*, 2004.
2. A.L. Brown. Constructing genome scale suffix trees. In *2nd Asia-Pacific Bioinformatics Conference*, 2004.
3. C.-F. Cheung, J.X. Yu, and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):90–105, 2005.
4. A.L. Delcher, A. Phillippy, J. Carlton, and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478–2483, 2002.
5. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. In *Workshop on Algorithm Engineering and Experiments*, 2005.
6. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. of the ACM*, 47(6):987–1011, 2000.
7. 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.
8. R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Software Practice & Experience*, 33(11):1035–1049, 2003.
9. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
10. 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.
11. E. Hunt, M.P. Atkinson, and R.W. Irving. A database index to large biological sequences. In *27th Int'l Conference on Very Large Data Bases*, 2001.
12. R. Japp. The top-compressed suffix tree: A disk-resident index for large sequences. In *BNCOD Bioinformatics Workshop*, 2004.
13. B. Phoophakdee and M. J. Zaki. Genome-scale disk-based suffix tree indexing. In *ACM SIGMOD Int'l Conference on Management of Data*, 2007.
14. K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.
15. Y. Tian, S. Tata, R.A. Hankins, and J.M. Patel. Practical methods for constructing suffix trees. *VLDB Journal*, 14(3):281–299, 2005.
16. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3), 1995.