A New Method for Indexing Genomes Using On-Disk Suffix Trees

Marina Barsky
Dept. of Computer Science
University of Victoria
BC, V8W 3P6, Canada
mgbarsky@cs.uvic.ca

Ulrike Stege
Dept. of Computer Science
University of Victoria
BC, V8W 3P6, Canada
stege@cs.uvic.ca

Alex Thomo
Dept. of Computer Science
University of Victoria
BC, V8W 3P6, Canada
thomo@cs.uvic.ca

Chris Upton
Dept. of Biochemistry & Microbiology
University of Victoria
BC, V8W 3P6, Canada
cupton@uvic.ca

ABSTRACT

We propose a new method to build persistent suffix trees for indexing the genomic data. Our algorithm DiGeST (**Di**sk-Based **Ge**nomic **S**uffix **T**ree) improves significantly over previous work in reducing the random access to the input string and performing only two passes over disk data. DiGeST is based on the two-phase multi-way merge sort paradigm using a concise binary representation of the DNA alphabet. Furthermore, our method scales to larger genomic data than managed before.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management—*Systems*; J.3 [**Computer Applications**]: Life and Medical Sciences

General Terms

Algorithms, Design, Performance

Keywords

suffix tree, disk structures, DNA indexing

1. INTRODUCTION

Since the Phi-X174 phage was sequenced in 1977, the DNA sequences of thousands of organisms have been decoded and stored in databases. Many biological advances heavily depended on the success of computational methods. However, genomic data is generated in vast quantities and the algorithms for exploring and analyzing such data are notorious for their high computational needs.

Large scale indexing of genomic data is a necessary ingredient in developing new methods to explore the molecular evolution of the species, to integrate cross-species genomic data, and to study gene structure, function, and regulation. In addition to the effective search for exact and approximate patterns, genome indexes can help discovering "ultra-conserved" regions in DNA of several species, as well as unique DNA markers which occur only in some species (cf. [33, 29]).

Since DNA cannot be effectively broken into words[1], the common text indexing methods, such as *inverted indexes* or *B-trees*, cannot be efficiently used. In [9], a new data structure, the *String B-tree*, was proposed a combination of the suffix tree and the B-tree. However, so far there is no practical method to build such a structure in external memory for large data sets.

Very well suited for the purposes of comparative genomics is a (generalized) suffix tree built on the available genomic data. If a suffix tree for multiple genomes could be efficiently built, then conserved regions would be easily "read" from the suffix tree, unique sequences would be found, and sequences of species would be efficiently compared.

Unfortunately, the size of the suffix trees is very large even for moderate genomic sequences, and thus the trees quickly outgrow the available main memory. For example, for an input string of 6 GB one would need at least 60 GB of RAM to hold its suffix tree. This calls for a disk-based method for building suffix trees.

The problem of efficiently building suffix trees in secondary storage for very large genomic data sets has recently drawn a lot of attention, cf. [13, 4, 2, 30, 31, 3, 8, 22, 23]. The most efficient methods, TDD and Trellis proposed in [31] and [22] respectively, scale up to the entire human genome – approximately 3 GB – resulting into a *persistent* suffix tree of a size in the tens of gigabytes (see [22]).

Unfortunately, being able to index a single genome is not good enough when it comes to perform comparisons of multiple genomes. Instead, one needs to build a common suffix tree for a *set* of DNA sequences (at least two). Often, the size of both such input and resulting suffix tree exceed what TDD and Trellis can handle. In their current form, both

[1]For example in the non-coding regions, which make about 95% of genomic DNA [33].

[31] and [22] are unable to efficiently handle inputs larger than 4 GB.

Can the algorithms of [31] and [22] be extended to handle such inputs? Although this might not be impossible, we point out (see Sections 3 and 5) some limitations which suggest that extending these methods would not scale well.

Thus, instead, we propose a new method to efficiently build persistent suffix trees for genomic datasets of larger size, allowing the indexing of more than one genome which is required for the envisioned applications. Specifically, we make the following contributions:

1. We present DiGeST[2], an efficient, secondary storage algorithm based on the well-known two-phase multiway merge sort paradigm. Notably, DiGeST performs only two passes on disk data, and has a very good locality of accesses to the input strings.

2. We present a new way for dividing the output suffix tree into subtrees of equal size. Previous methods, like TDD and Trellis divide the output in pieces which can be significantly unbalanced. In contrast, our method, based on the lexicographical ordering of suffixes, guarantees subtrees of equal sizes.

3. We show that DiGeST easily scales for larger inputs than before. For example, DiGeST is able to handle with ease real DNA inputs of size 6 GB and to build suffix trees of size in the hundreds of gigabytes in just four hours, which TDD and Trellis+ (an improved version introduced in [23]) are unable to handle.

4. We show that, for other large inputs, which TDD and Trellis+ can handle, DiGeST significantly outperforms them. We present an experimental analysis and discussion of the performance of DiGeST versus Trellis+, which, as the best algorithm known so far, we use for our benchmarking.

The rest of the paper is organized as follows. Section 2 describes the background and terminology on strings and suffix trees. Section 3 reviews the related work, highlighting the so far best methods TDD and Trellis. Section 4 describes our new algorithm DiGeST, which is then compared to Trellis+ in Section 5. Section 6 concludes with final remarks.

2. BACKGROUND

Let an *input string* S be a sequence of N consecutive characters from an alphabet Σ. We denote by $S[i, j]$, where $0 \leq i < j \leq N$, the substring of S starting at position i and ending at position j. The suffix of S starting at position i is the substring $S[i, N]$. A *suffix tree* for string S of size N is a rooted directed tree with exactly N leaves. The key feature of a suffix tree is that each suffix $S[i, N]$ of S is represented by a path from the root to a leaf node L_i.

[2]DiGeST is an anagram of the phrase:
Disk-Based **Ge**nomic **S**uffix **T**ree. The program is available at: *http://webhome.cs.uvic.ca/∼mgbarsky/digest/*

Each internal node other than the root has at least two children and each edge is labeled with a corresponding substring from S. A suffix tree is a compact index of all distinct substrings of a given string with each edge representing one such distinct substring.

An example of a suffix tree for input string $S = abbabbaa$ is depicted in Figure 1. Any substring of S can be located starting from the root by traversing the edges according to their labels.

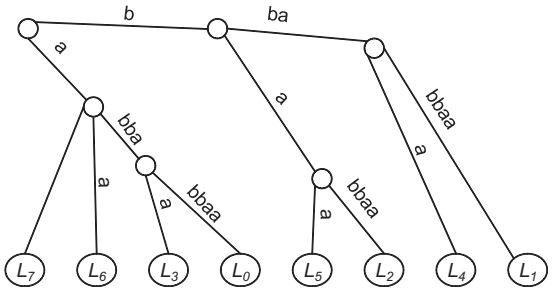

Figure 1: A suffix tree for input string $S = abbabbaa$. In order to find all occurrences of query string abb in S, match abb starting from the root and then reaching to the least common ancestor of the leaves L_0 and L_3. The positions specified by these leaves indicate the start of occurrences of the query string in S.

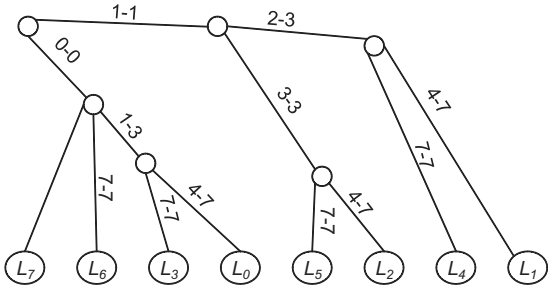

Figure 2: Compressed suffix tree for string $S = abbabbaa$. In order to find all occurrences of query string $bbabb$, traverse the tree matching at positions 0,1, and 3 of the query string, then retrieve L_1 and perform character-by-character verification of the query. Note, that for query string $bbaba$ the tree traversal is identical, but after verification against the suffix of S starting at position 1 we find that the query string does not occur in S.

Note that, if we label each edge with the actual characters of S, the size of the resulting tree is $O(N^2)$, which is prohibitive for most real-life applications. Such a tree is called *uncompressed suffix tree* and is mainly of theoretical interest. In practice, all methods build *compressed suffix trees.*

A compressed suffix tree does not store explicitly the labels of the edges. The edge labels are represented by an ordered pair of integers denoting its start and end positions in the input string. The compressed suffix tree for the input string above is shown in Figure 2.

Note that, to search for a query string q in a compressed suffix tree, we could (naively) compare the characters of q to the characters of S as indicated by the positions of the edge labels. This type of search, unfortunately, requires multiple random accesses to the input string, and this is quite inefficient when S is large. So, is it really worth to build a compressed suffix tree?

Notably, massive random access to S during a search can be avoided by performing a "blind search" as suggested in [9]. Observe that the outgoing edges from an internal node are indexed according to the character specified by their start position. Only these (implied) first characters of the traversed edges are in fact matched against corresponding characters in q. Thus we "jump" in q by the lengths of the traversed edges. If matching q (in this way) fails, we conclude that q is not a substring of S. Otherwise, if q matches some path π in the tree, we retrieve a leaf L_i from the subtree induced by π and perform a follow up validation of q against $S[i, i + |q|]$. Note that this requires at most one random access to string S per query.

Our approach for building suffix trees. In this paper, we focus on *generalized suffix trees* (see [11]) which index more than one input string. In this case, all suffixes of each of the input strings are inserted into the tree. In comparison to suffix trees for one input string some leaves may represent multiple suffixes belonging to different input strings. Thus, the leaves now store possibly more than one start position along with input string identifiers. An example of a generalized suffix tree for two input strings is shown in Figure 3. Note, that now more information should be stored at each edge label, namely the particular input string identifier.

In our approach, we perform only sequential reads and writes of disk data. Specifically, we carry out a special two-phase multi-way merge sort (cf. [10]), but instead of sorting records we sort suffixes of the input string. The suffixes are represented only by their start positions. The input string is first divided into partitions and then their suffixes are sorted in lexicographical order using an efficient algorithm by Larsson [19]. The result of this sorting phase is a set of *suffix arrays* [20], each being a list of start positions of sorted suffixes.

In our merge phase, consecutive pieces of each of the suffix arrays are read from the disk into input buffers. A "competition" is run among the top elements of each buffer and the "winning" suffix migrates to an output buffer organized as a suffix tree. When the output buffer is full we empty it to disk.

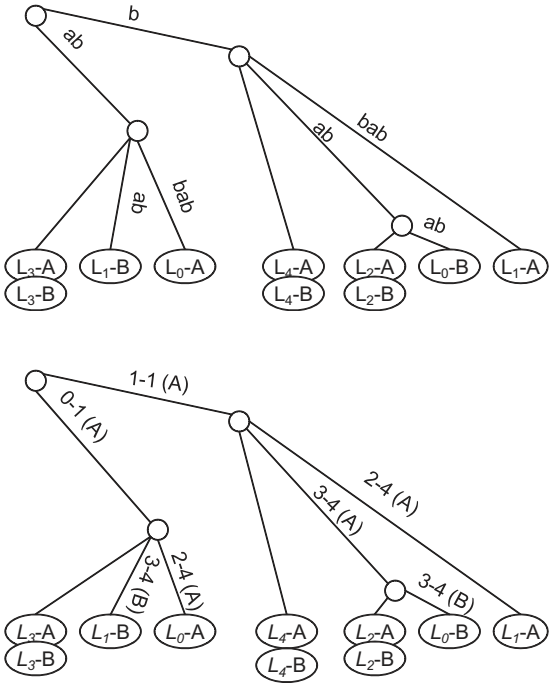

Figure 3: Example of generalized suffix trees for input strings $A = abbab$ and $B = babab$. Queries of type "what is the longest substring common to A and B" can be solved in linear time. An answer to this query is the path corresponding to substring bab, common to the suffixes starting at position 2 in A and B.

3. RELATED WORK

Several approaches for disk-based construction of suffix trees are considered in the literature, cf. [13, 4, 2, 30, 3, 8, 31, 23]. The most recent and scalable methods are [31] and [22] which we describe in the following.

TDD. A new chapter in building suffix trees on disk was started by the research of Tata et al. [30, 31]. Their top-down disk based technique (TDD) significantly reduces, although not completely avoids, massive random access to the suffix tree being built.

However, TDD accesses the input string randomly and this degrades the performance when the string is large. TDD also degrades when the input data is skewed. This is because TDD partitions the input according to prefixes of equal length. For real life data, the size of partitions can be quite different and consequently there may exist large subtrees causing memory overflow.

TDD combines the top-down technique with the prefix-based partitioning of the input. The problem is that the total number of different prefixes grows exponentially with the prefix length. For example, a prefix length of 7, used by authors to process 3 GB of the human genome, causes 16,384 independent suffix trees to be built. For building each one of these trees, the entire input string is randomly accessed. Thus, this partitioning technique cannot scale to handle bigger inputs as for instance several eucariotic genomes.

Trellis. Phoophakdee and Zaki in [22] propose a new method for building suffix trees. TRELLIS is a partition and merge strategy. It breaks the input string into several small partitions and builds the suffix tree for each partition using the in-memory linear time algorithm by Ukkonen [32]. Each such suffix tree is written to disk. After this first phase, a collection of variable-length prefixes is created, addressing

the data skew problem. Then for each prefix p in the collection, TRELLIS loads into main memory the subtrees (of the previously built trees) for prefix p. These subtrees are then merged into one subtree containing all the suffixes sharing p as a prefix.

In [23], an improved version of TRELLIS, namely TRELLIS+ was introduced which is based on the same principles as TRELLIS, but uses fewer and larger partitions resulting in less tree merge operations.

In the merge phase, edges of subtrees are compared character-by-character according to the positions on each edge label. Since the trees are compressed, massive random access to the input string is performed.

The tree merge phase performs multiple random disk reads of the trees built in the previous phase. In fact, a tree of each partition (substring) is accessed on disk as many times as the number of variable-length prefixes. This implies that these random I/Os will cause significant performance degradation for larger inputs, when the total number of prefixes and the total number of partitions grows. The total number of random disk reads in the merge phase is proportional to the number of prefixes multiplied by number of partitions, and depends, therefore, on the square of the input length.

Despite using variable length prefixes in order to balance the sizes of the resulting subtrees, we observed that these subtrees are not completely balanced as some are very small while others are an order of magnitude larger.

TRELLIS also contains a post-processing step for the suffix links recovery. In our opinion, the suffix links in the on-disk tree have a limited usage, since they point in most cases to different suffix sub-trees, layered in distant disk locations. This means that an assumed constant-time jump following suffix link causes in fact an entire random disk access. For example, the streaming algorithm for finding all maximal unique matches between two genomic sequences as a part of the MUMmer program for alignment genomes [7, 18], uses suffix links to stream the second genome against the suffix tree built for the first genome. If the sub-trees are on disk, following the suffix link causes another sub-tree to be uploaded from disk, which leads to the same number of disk reads as if this new sub-tree was traversed from the root.

To summarize, both TDD and TRELLIS incur a great number of random disk I/Os which are heavily felt when trying to process large inputs.

As it was shown in [22], TRELLIS significantly outperforms TDD for input sizes larger than 1 GB, thus being the fastest known method for building persistent suffix trees. Therefore, we use TRELLIS+, as a benchmark for evaluating our method.

4. BUILDING SUFFIX TREES ON DISK

For simplicity, in the following we work with only one input string, but clearly explain in the appropriate places how some particular step of our method is extended to multiple input strings.

Our method for building a suffix tree on disk consists of the following main components:

1. **Preprocessing.** We create, from the original input string, a small number, say k, of input files of approximately the same size. In this step we also encode the input string (DNA) as a sequence of bits with two bits per character.

2. **Sorting of suffixes.** We sort suffixes in each partition by using Larsson's algorithm [19]. In addition to the output of this algorithm, we also attach to each suffix start position a short prefix of the suffix. These prefixes significantly improve the performance of the merging phase. At the end of this phase, for each partition, we have on disk a suffix array with attached prefixes.

3. **Merging of prefix-attached suffix arrays.** Consecutive pieces of each of the k suffix arrays are read from the disk into input buffers. A "competition" is run among the top elements of each buffer and the "winning" suffix migrates to an output buffer organized as suffix tree. When the output buffer is full we empty it to disk.

These components are illustrated in Figure 4.

We further describe our method in more details.

4.1 Preprocessing of the input

The size of partitions is determined based on the amount of available memory. To build a suffix array for each partition using Larsson's algorithm [19], we need 8× *partition size* bytes. In addition we need an output buffer to collect the suffix positions along with 64-bit long prefixes attached. We found that we can quickly process partitions of size 100 MB. Note, that even though we could fit more into the available memory, the Larsson's algorithm involves random accesses to the input string, and thus, its performance degrades for bigger partition sizes due to cache misses. After choosing the size of the partitions, we determine their number k.

We partition an input string into consecutive substrings of size N/k. For a technical reason to become clear soon, we add a "tail" to each partition (except the last). Let P_i and P_{i+1} be two consecutive partitions. We attach a prefix t of P_{i+1} as a tail to P_i. Prefix t is the shortest prefix of P_{i+1} such that it does not occur anywhere as a substring of P_i. For random and real DNA sequences, the length of t was never more than 1000.

For more than one input string (for example, a set of human chromosomes), we partition only the input strings of size greater than 100 MB, leaving the rest of the input strings in their separate files.

4.2 Sorting of suffixes

The sorting of suffixes in each partition is performed by Larsson's *quicksufsort* algorithm [19], using the implementation from [34].

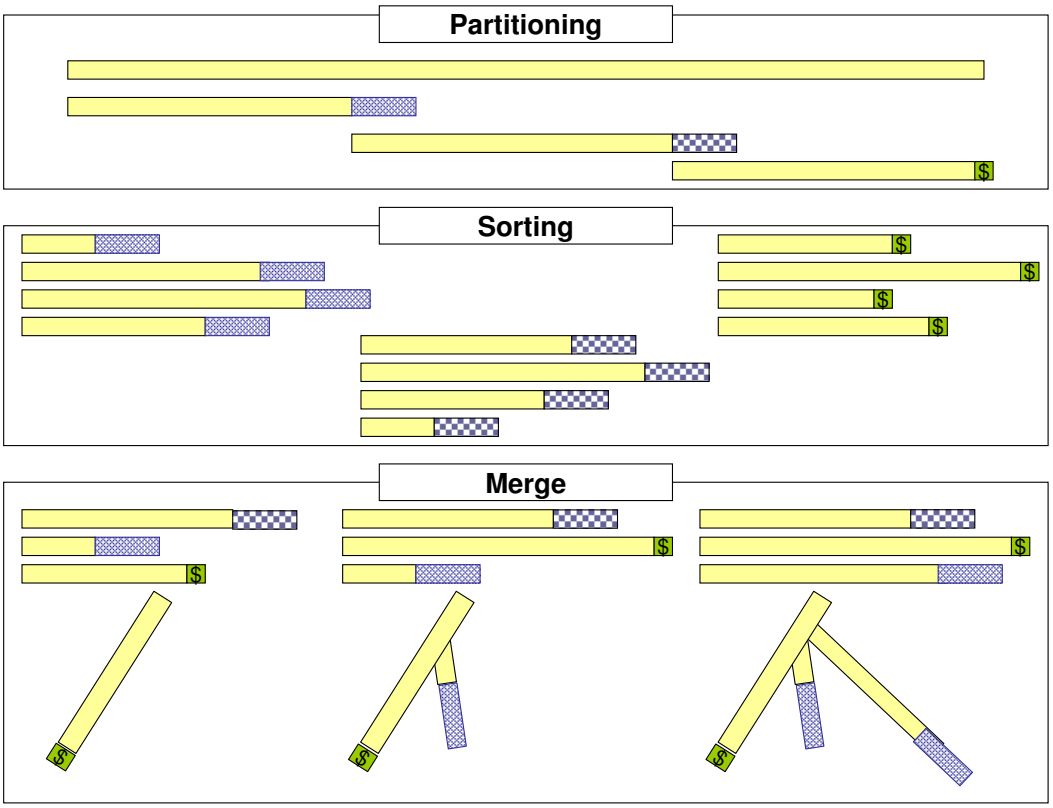

Figure 4: Overview of our method. Textured patterns at the end of each suffix correspond to tails, added to preserve the correct lexicographical order of the suffixes. The detailed description of each step is presented in Sections 4.1, 4.2, and 4.3 respectively.

Larsson improves the practical behavior of Manber-Myers algorithm [20] by avoiding the scanning of the entire array in each of the algorithm's $logN$ passes and using a ternary-split Quicksort as a sorting subroutine. Our choice in using this algorithm for sorting suffixes was influenced by the experimental results of [25]. Note that one can use any efficient suffix sorting algorithm for main memory (cf. [21, 16, 14, 15, 12]) in the first phase. We were interested in optimizing the method for disk accesses rather than improving running time of in-memory sub-routines.

While sorting the suffixes of the partition, we need to guarantee that the sort is consistent with the lexicographical order of suffixes in the whole input string. For this, recall that, in the preprocessing phase, when an input file is broken into several partitions, a tail t was attached to the end of each partition – the prefix of the next partition which does not occur anywhere else in the current partition. We next prove that this is necessary and sufficient to ensure that all suffixes in the partition are in the correct order.

PROPOSITION 1. *Let*

1. *P be a partition of the input string S*
2. *t be the tail appended to P*
3. *p_i, p_j be two suffixes of P starting at (global) positions i and j, respectively,*
4. *s_i, s_j be the suffixes of S starting at positions i and j, respectively.*

Then, the concatenation $p_i \cdot t \leq_{lex} p_j \cdot t$ if and only if $s_i \leq_{lex} s_j$.

Proof. *Only if.* Straightforward.

If. Without loss of generality suppose that $i < j$. Since t is not a substring of P, $p_j \cdot t$ cannot be a prefix of $p_i \cdot t$.

As such, let c_{i+k} and c_{j+k} be the first characters in $p_i \cdot t$ and $p_j \cdot t$, respectively, where $p_i \cdot t$ and $p_j \cdot t$ differ. Clearly, if $c_{i+k} <_{lex} c_{j+k}$ ($c_{i+k} >_{lex} c_{j+k}$) then $s_i <_{lex} s_j$ ($s_i >_{lex} s_j$). □

Note that if a tail t cannot be found, we cannot guarantee a correct sorting of suffixes in each partition. However, in practice, we have not yet encountered such a case.

Once the suffixes are sorted, we write their start positions to disk. For each partition, we have a list of suffix start positions. The order of these positions is according to the lexicographical order of their corresponding suffixes. Further, next to each start position, we store the 32 character (64 bits) prefix of the suffix in the form of two four-byte numbers. This is possible because the alphabet of DNA has only four letters, and thus, each letter can be compactly represented by two bits.

In the rest of the paper, for simplicity we blur the distinction between a suffix and its starting position.

4.3 Merging of prefix-attached suffix arrays

Merging of suffix arrays into a suffix tree incurs multiple random accesses to the input string. In order to increase the amount of the input that can fit in the available main memory we encode each distinct letter of the meaningful DNA alphabet $\{a, c, g, t\}$ as a 2-bit string. The same encoding was used by TRELLIS+, the program we compare our results to.

Thus, we consider $\Sigma = \{0, 1\}$, and use a special symbol \$ for denoting the end of a string. Note that a string over any alphabet can always be reduced to the binary alphabet by representing each character as a sequence of bits and then concatenating these binary sequences.[3]

We set $M = 2N$ for the length of the DNA input string coded in binary as above.

Our merge is as follows. We use one input buffer for each of the k sorted lists of suffixes. The input buffers are loaded with suffixes from the corresponding lists. Then a competition is run among the top elements of each input buffer. The winning element migrates to an output buffer organized as a suffix tree.

For the competition we use a priority queue implemented as a heap of size k (number of partitions). To determine the relative order of the suffixes from different lists, we first compare their attached prefixes, stored as two long integers. Only if both of them are equal, we access the input string at the corresponding positions. We found that, this only happens in practice for a very small fraction of the suffixes, approximately 2.5% in real DNA sequences and basically never for synthetic DNA sequences. Without storing such prefixes, different suffix comparisons would cause multiple random accesses to the input string, which is not desirable due to excessive cache misses.

The smallest element removed from the heap is added to a growing suffix tree in the output buffer. A naive method of adding a new suffix into a suffix tree involves comparing the characters of the new suffix to the corresponding edges of the tree starting from the root. Since we only have positions rather than characters, this would involve massive random access to the input string. Thus, in order to avoid these comparisons, we first find the length LCP of the longest common prefix between the last added suffix, say s_1, and the next suffix, say s_2, to be added. The prefixes attached to the elements of the heap help us determine the LCP of these two suffixes without accessing the input string in the vast majority of the cases.

Once we know the LCP of s_1 and s_2, we can add the leaf corresponding to s_2 by traversing the lexicographically largest path in the existing tree up to LCP characters and creating a new internal node and a new leaf.

In the following we describe our merge algorithm in more details.

The growing tree is represented as an array of nodes. This array fills an output buffer of a pre-calculated size which is then flushed to disk. The lexicographically largest suffix in this tree, is added to a collection of "dividers" which serve locating multiple trees on disk. We call the path in the suffix tree corresponding to the lexicographically largest suffix the *boundary path*.

At the end of the merging phase, we have on disk a forest of $M/outputBufferSize$ suffix trees as well as a collection of the same number of dividers corresponding to the boundary path of each tree.

All trees are of equal size, and thus, the problem of data skew is now completely avoided.

Let T_i be one of the suffix trees obtained by the algorithm. The leaves of tree T_i correspond to suffixes of a certain lexicographical range as captured by dividers d_{i-1} and d_i.

Further, each tree is small enough to be quickly loaded into the main memory to perform search or comparative analysis. Since the number of dividers is small, and we store only their first 64 bit prefixes, they can be loaded entirely into the main memory. For example, for the human genome the number of dividers is approximately 6,500.

Descriptive pseudocode of the suffix merge

1. Create k input buffers of size *buf* each.
2. Fill these k buffers with suffixes from the corresponding k sorted suffix arrays.
3. Initialize a heap H with the first suffixes of each of the k input buffers.

[3]We note that in real DNA sequences there are unidentified (or "unknown") characters denoted by n, which does not belong to the $\{a, c, g, t\}$ alphabet. For example, human chromosome 22 contains a large block of of such symbols at its beginning. We discard these characters from our binary encoded input. Trellis also discards such characters from the input. We remark that this does not create a problem when using suffix trees. For this, one can record the cut positions and easily map the characters in the transformed string to characters in the original string.

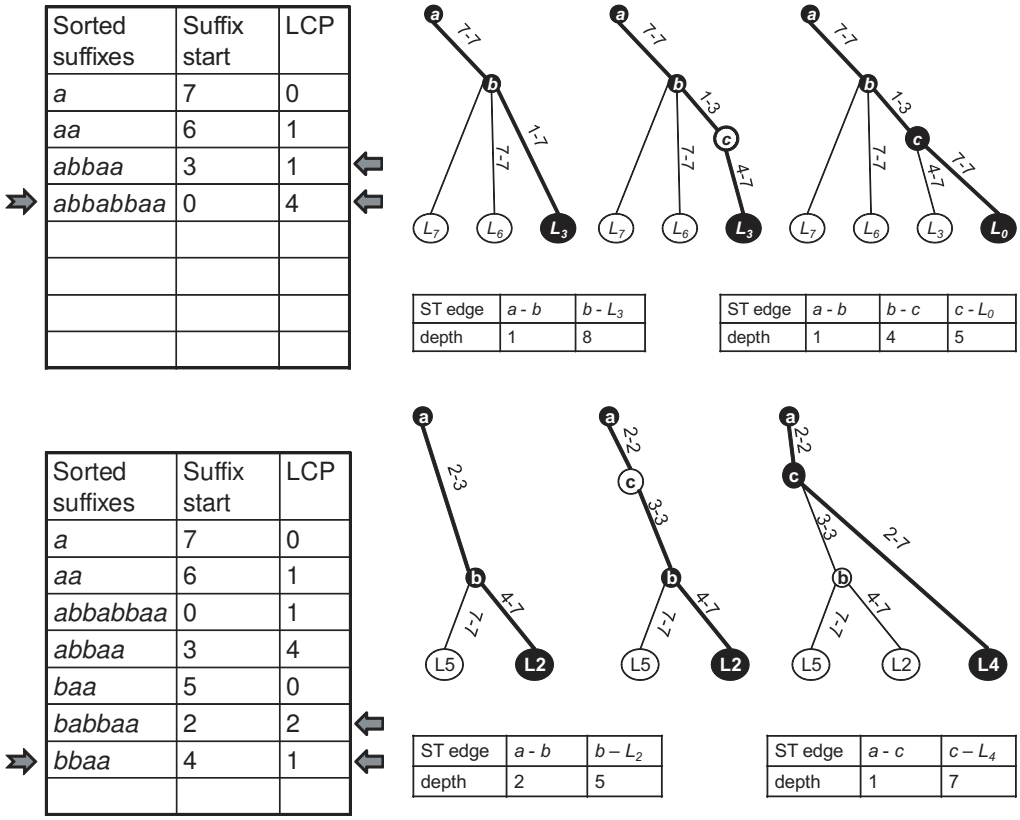

Figure 5: Two examples of adding a suffix for input string $abbabbaa$. [Left] Adding the suffix starting at position 0. In this case, the LCP of this suffix with the previous suffix, starting at position 3, is larger than the LCP between the latter and the suffix starting at position 6. The last edge bL_3 on the boundary path (abL_3) splits, and new internal node c and leaf L_0 are created. [Right] Adding the suffix starting at position 4. Here, the LCP of this suffix with the previous suffix, starting at position 2, is smaller than the LCP between the latter and the suffix starting at position 5. In this case, we first need to locate the corresponding edge ab on the boundary path abL_2 (for example using binary search). Then edge ab splits and new internal node c and leaf L_4 are created. The sorted suffixes are shown in the tables left of the trees. The boundary path arrays are shown below the trees.

4. Remove the top element (the lexicographically smallest suffix) from H. Let this top element belong to partition P_i.
5. Initialize the suffix tree of the output buffer by creating one leaf node for this first suffix. Store this suffix and its 64-bit prefix in a variable *lastAdded*.
6. Insert into H the next suffix from the buffer corresponding to P_i.
7. Remove top element *curr* from H.
8. Find the length LCP of the longest common prefix of *curr* and *lastAdded*. Further record the bit of *curr* in position $curr+LCP+1$. Call this bit *firstBitAfterLCP*.
9. Traverse the suffix tree starting from the root and following the boundary path continuing until depth LCP. At depth LCP split the edge reached, creating a new internal node ν. Create a new leaf child of ν representing the rest of suffix *curr*.
10. If the output buffer is full, then
    (a) flush it to disk,
    (b) store the 64-bit prefix of *curr* and a reference to the flushed tree in array *dividers*,
    (c) go to step 4.

    Otherwise, go to step 6.

In the next proposition we show that adding new nodes to the growing suffix tree by the procedure of step 9 is correct, i.e. that the split point for each next suffix cannot be found anywhere except on the boundary path of the tree built so far.

PROPOSITION 2. *Let*

1. *T be a suffix tree currently being built in the output buffer,*
2. *s_1 be the suffix last added to T,*

3. s_2 be the suffix to be added next into T in step 9.

Then, split point ν (the parent node for leaf s_2) lies on the boundary path of T.

Proof. There does not exist a suffix s_3 such that $S[s_1, j] <_{lex} S[s_3, j] <_{lex} S[s_2, j]$ for any $1 \leq j \leq N$. This is true because otherwise s_3 would be the next suffix to be inserted after s_1.

Since s_1 corresponds to the (lexicographically) greatest suffix added to T, the boundary path of T corresponds to $S[s_1, N]$, and thus, covers all the prefixes of s_1 including substring $S[s_1, s_1 + LCP]$. Hence, in order to locate substring $S[s_1, s_1 + LCP] =_{lex} S[s_2, s_2 + LCP]$, we need to follow the boundary path of T. Therefore split point ν lies on the boundary path. □

The next proposition shows how we make use of the binary alphabet during step 9 of the above merge procedure, without additional random access to the input string. Note that each node has exactly two children, namely the 0-child and 1-child.

PROPOSITION 3. *Let*

1. *T be a suffix tree currently being built in the output buffer,*
2. *s_1 be the suffix last added to T,*
3. *s_2 be the suffix to be added next into T in step 9.*

Then the insertion of s_2 splits an existing edge such that it creates the 1-child leading to the leaf s_2.

Proof. The suffix starting at s_2 has a common prefix of length LCP with the suffix starting at s_1. Then the character $S[s_2 + LCP + 1]$ is greater than $S[s_1 + LCP + 1]$. Since our alphabet is binary, we have $S[s_2 + LCP + 1] = 1$. □

The only case a new 0-child can lead to the leaf corresponding to suffix s_2 is when $s_1 + LCP = M$. That is, suffix s_1 is a prefix of suffix s_2 and the leaf corresponding to s_2 will be a child of the node corresponding to s_1, which is transformed from a leaf to an internal node.

Note that, if using $\{a, c, g, t\}$ as alphabet instead of $\{0, 1\}$, then it is not possible to build the suffix tree as described above without incurring random accesses to input string S. This is because when splitting an edge for adding a leaf corresponding to s_2, we would have needed to check the character $S[s_1 + LCP + 1]$.

Several cases of adding a suffix to a growing suffix tree are shown in Figure 5.

Remarks

1. With our method, the total number of suffixes remains N (not $M = 2N$) as we sort only N suffixes (in the sorting phase) and merge these suffixes creating one leaf for each suffix.
2. Any internal node in a suffix tree has at least two children. As we use the binary alphabet, the number of children is limited to be at most two. This allows using two child pointers only (per node) and a constant time access to these children.

Finally, our algorithm performs only two passes on the disk data, namely two reads and two writes. The random access to the tree being built is now completely avoided.

Notably, exactly the same merge algorithm can be used for creating on-disk suffix arrays, since each next suffix added to the tree is in lexicographical order.

4.4 Properties of the output

Let us now look at the on-disk suffix tree forest produced by DiGeST. It is a collection of suffix trees each of which is of such a size that it can be quickly loaded into main memory by one sequential disk read. Each tree corresponds to some lexicographical interval. The collection of minimum and maximum prefixes of constant length (dividers) for each tree can be kept entirely in main memory due to its small size.

Searching for an exact pattern is easy with this layout of the suffix trees on disk. First, we locate the lexicographical range of the pattern by finding the corresponding divider (in main memory). Next we load into memory the suffix tree corresponding to this divider. We perform the blind search for the exact pattern described in detail in Section 2. Note that this search requires not more than two random disk accesses - one for uploading the tree and one for verification of pattern against one substring of the input string (in case that the input string resides on disk).

When the depth-first traversal of the entire suffix tree is required, for example in the case of finding common substrings for the set of input strings, the consecutive trees are loaded and traversed in main memory. This is performed by sequential disk reads.

The efficiency of other algorithms for this on-disk suffix tree layouts is yet to be investigated.

5. EXPERIMENTAL RESULTS

In this section, we present the performance evaluation of DiGeST in comparison with Trellis+, which is the best algorithm known so far for building suffix trees on disk. The source code of Trellis+ was obtained from [36]. DiGeST was implemented in C and was compiled with GNU gcc compiler, version 4.1.2. All experiments were performed on a machine with an Intel Core Duo 2.66 Ghz CPU, 2 GB RAM and 4MB L2 cache under Ubuntu 7.04, 32-bit Linux. Both programs were compared for different input lengths with the

Data set	Chromosomes of			Total Size, GB
	Human	Chimpanzee	Zebrafish	
1	1,2,20,21	1,2,20,21		0.9
2	2 - 6	2 - 6		1.8
3	1 - 8	1 - 8		2.7
4	1 - 12	1 - 12		3.6
5	1 - 18	1 - 18		4.5
6	1 - 23	1 - 23		5.4
7	1 - 23	1 - 23	1 - 25	6.3

Table 1: Datasets used in our experiments. For example, line 1 of the table says that dataset 1 consists of sequences of chromosomes 1, 2, 20 and 21 of both human and chimpanzee, with the total input length of approximately 0.9 GB. Each chromosome was stored in a separate input file. Their total size is shown in the last column of the table. The last line represents a data set generated from three entire genomes of human, chimpanzee and zebrafish.

same amount of available main memory (namely, 2 GB). Recall that all recent algorithms, including TDD, TRELLIS and DIGEST, to be efficient require that the input string resides in main memory. The 2-bit per character compression of the input made it possible to build the suffix tree for the input string of 6 GB with 2 GB of total RAM. This tree is of size approximately 180 GB. In practice, RAM cannot be extended to such sizes to build the suffix tree entirely in main memory.

Data Sets	1 (0.9GB)	2 (1.8GB)	3 (2.7GB)	4 (3.6GB)	5 (4.5GB)	6 (5.4GB)	7(6.3GB)
Trellis	46	92	148	216			
DiGeST	34	68	102	135	174	208	244

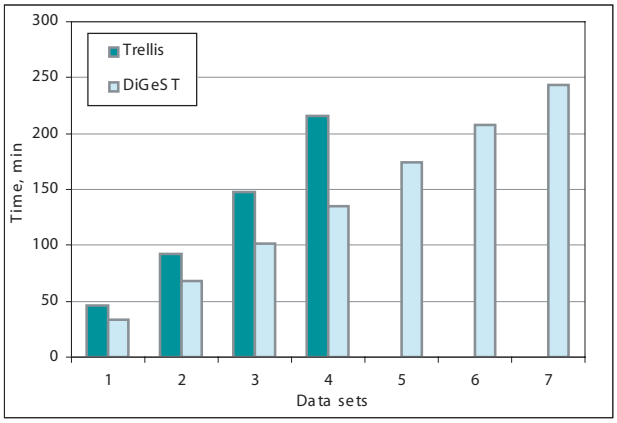

Figure 6: Running times (in minutes) for real DNA. Observe that, for real DNA inputs of total size 3.5 GB DIGEST outperforms TRELLIS+ by about 40%. Furthermore, for larger sizes there is only DIGEST that is able to produce results.

First, we evalaluated the performance of DIGEST versus TRELLIS+ for the human genome which is about 3 GB. DIGEST was able to build the suffix tree in 1.5 hours, while TRELLIS took 2.5 hours.

In the following, we present further results demonstrating

Data Sets	1 (0.9GB)	2 (1.8GB)	3 (2.7GB)	4 (3.6GB)	5 (4.5GB)	6 (5.4GB)	7(6.3GB)
Trellis	39	79	126	191			
DiGeST	33	54	86	118	155	188	229

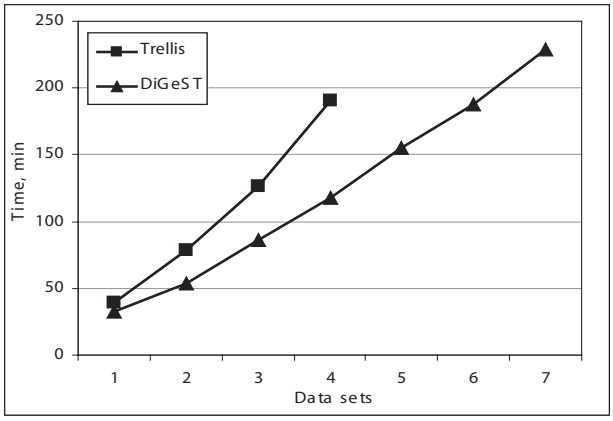

Figure 7: Running times (in minutes) for synthetic DNA. Similar to the previous figure, it is only DIGEST that is able to produce results for sequences larger that 4 GB.

the performance of DIGEST. Namely, we evaluated the performance of DIGEST versus TRELLIS for the following DNA types:

1. Collection of the corresponding chromosome pairs of the genomes of several species, namely human, chimpanzee, and zebrafish (obtained from [35]). We grouped the corresponding chromosome pairs into inputs of approximate total size from 1 GB to 6 GB. We believe that such combinations correspond to the goal of comparative genomics.
2. Synthetic DNA sequences which we generated using a uniform random distribution of characters.

The details of the input datasets for type 1 are presented in Table 1.

The running times for DIGEST and TRELLIS+ are given in Figure 6 and Figure 7 for the data sets of Table 1 and synthetic DNA sequences, respectively.

Observe that DIGEST significantly outperforms TRELLIS+ for inputs greater than 1 GB. For example, for real DNA inputs of total size 3.5 GB DIGEST outperforms TRELLIS+ by about 40%.

Our gain in performance is due to the fact that DIGEST performs mostly sequential disk I/Os, whereas TRELLIS+, in its tree merge phase, performs multiple random reads of the trees built in the previous phase.

We believe that, for inputs of size greater than 4 GB, the number of tree merges of TRELLIS+ , and the number of partition trees to load for each merge, will cause a great deal of random I/Os.

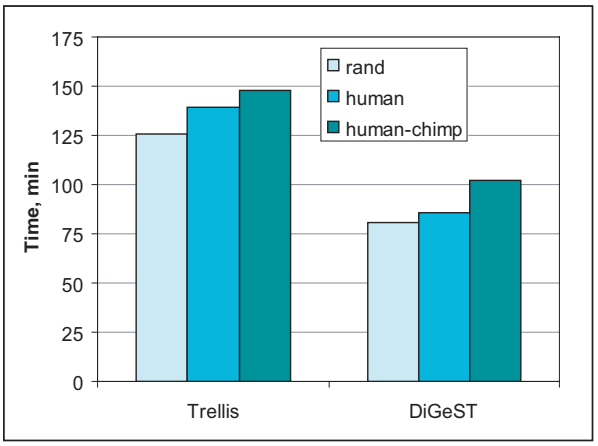

Figure 8: Running times (in minutes) for Trellis+ and DiGeST on three different inputs of size approximately 3 GB each.

On the other hand, DiGeST scales because it never reads the same piece of disk data more than once, and writes the suffix trees corresponding to the lexicographically partitioned suffixes performing only one random disk access per tree.

Next we study the behavior of DiGeST and Trellis+ with respect to the type of input. In Figure 8, we fix an input size of approximately 3 GB which is about the size of the human genome and plot the results of DiGeST and Trellis+ for the three types of data. Namely, we consider synthetic DNA with uniform distribution of characters, human genome, and two genome subsets of similar species, human and chimpanzee. Human genome has more and longer repetitions than synthetic DNA. On the other hand, the human and chimpanzee genomes are quite similar and have even more and longer common substrings.

Both DiGeST and Trellis+ perform slightly better on synthetic DNA than on the other data sets used in the experiments.

6. CONCLUSIONS

We presented DiGeST, a new algorithm for indexing large DNA sequences using generalized suffix trees in secondary storage. DiGeST significantly improves over the former best method, Trellis+. Our algorithm is the first one able to index more than one genome at a time.

7. REFERENCES

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1): 53–86, 2004.

[2] S.J. Bedathur and J.R. Haritsa Engineering a fast online persistent suffix tree construction. *20th Intl. Conf. on Data Engineering*, 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] R. Clifford, and M.J. Sergot Distributed and paged suffix trees for large genetic databases. *Proc. of 14th Symposium on Combinatorial Pattern Matching*: 70–82, 2003.

[5] A. Crauser and P. Ferragina A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory. *Algoritmica*, 32(1): 1–35, 2002.

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

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

[8] R. Dementiev, J. Krkkinen, J. Mehnert, and P. Sanders Better external memory suffix array construction. *Proc. of the 7th Workshop on Algorithm Engineering and Experiments*, 2005.

[9] P. Ferragina and R. Grossi The string B-tree: a new data structure for string search in external memory and its applications. *J. of the ACM*, 46(2): 1999.

[10] H. Garcia-Molina, J.D. Ullman, J.D. Widom Database System Implementation. Prentice-Hall, Inc, 1999.

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

[12] W-K. Hon, K. Sadakane, and W.-K. Sung Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *Proc. of the 44th Annual IEEE Symposium on Foundations of Computer Science*: 251, 2003

[13] E. Hunt, M.P. Atkinson, and R.W. Irving A database index to large biological sequences. *The VLDB Jornal*, 7(3): 139–148, 2001.

[14] J. Kärkkäinen, and P. Sanders Simple linear work suffix array construction. *Proc. 13th Int. Conf. on Automata, Languages and Programming*, 2003

[15] D.K. Kim, J.S. Sim, H. Park, and K. Park Linear-time construction of suffix arrays: (Extended abstract). *Proc. of CPM Conf.*, LNCS 2676: 189–199, 2003

[16] P. Ko and S. Aluru Space efficient linear time construction of suffix arrays. *J. of Discrete Algorithms*, 3 (2-4): 143–156, 2005

[17] S. Kurtz Reducing Space Requirement of Suffix Trees. *Software Practice and Experience*, 29(13): 1149–1171, 1999.

[18] S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S.L. Salzberg Versatile and open software for comparing large genomes. *Genome Biology*, 5(R12): 2004.

[19] N.J. Larsson and K. Sadakane Faster suffix sorting. Tech. Rep. LUCS-TR:99-214 of the Dept. of Comp. Sc., Lund University, Sweden, 1999.

[20] U. Manber and E. Myers Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. of*

Computing, 22(5):935–948, 1993.

[21] G. Manzini and P. Ferragina Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40: 33–50, 2004.

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

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

[24] E. Pennisi DNA Study Forces Rethink of What It Means to Be a Gene. *Science*, 316 (5831): 1556–7, 2007

[25] S.J. Puglisi, W.F. Smyth and A.H. Turpin A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.,* 39(2): 2007.

[26] T. Ryan Gregory The evolution of the genome. Academic Press, 2005.

[27] K. Schurmann, J. Stoye Suffix-tree construction and storage with limited main memory. *Tech. Rep. 2003-06 of the University of Bielefeld, Germany*, 2003.

[28] R. Sinha, S.J. Puglisi, A. Moffat, and A. Turpin Improving suffix array locality for fast pattern matching on disk. *Proc. of 2008 SIGMOD Conf.*, 661–672, 2008.

[29] A. Stark et al. Discovery of functional elements in 12 Drosophila genomes using evolutionary signatures. *Nature*, 450 : 219–232, 2007.

[30] S. Tata, R.A. Hankins, and J.M. Patel Practical suffix tree construction. *Proc. of 30th VLDB Conf.*, 36–47, 2004

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

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

[33] A. Woolfe et al. Highly conserved non-coding sequences are associated with vertebrate development. *PLoS Biology*, 3(1), e7 doi:10.1371/journal.pbio.0030007

[34] Strings, Compression, and Orchestra: *www.larsson.dogma.net/research.html*

[35] USCS Genome Browser: *hgdownload.cse.ucsc.edu/downloads.html*

[36] Benjarath Pupacdi's Home Page: *www.cs.rpi.edu/∼zaki/software/trellis*