# Suffix Trees for Inputs Larger than Main Memory

Marina Barsky, Ulrike Stege and Alex Thomo

University of Victoria
BC, V8W 3P6, Canada

**Abstract.** A suffix tree is a fundamental data structure for string searching algorithms. Unfortunately, when it comes to the use of suffix trees in real-life applications, the current methods for constructing suffix trees do not scale for large inputs. As suffix trees are larger than the input sequences and quickly outgrow the main memory, the first attempts at building large suffix trees focused on algorithms which avoid massive random access to the trees being built. However, all the existing practical algorithms perform random access to the input string, thus requiring in essence that the input be small enough to be kept in main memory. The constantly growing pool of string data, especially biological sequences, requires us to build suffix trees for much larger strings.

We are the first to present an algorithm which is able to construct suffix trees for input sequences significantly larger than the size of the available main memory. Both the input string and the suffix tree are kept on disk and the algorithm is designed to avoid multiple random I/Os to both of them[1]. As a proof of concept, we show that our method allows to build the suffix tree for 12GB of *real* DNA sequences in 26 hours on a single machine with 2GB of RAM. This input is *four times* the size of the Human Genome, and the construction of suffix trees for inputs of such magnitude was never reported before.

## 1 Introduction

Nowadays, textual databases are among the most rapidly growing collections of data. One of these collections, the collections of sequenced genomes, is a textual database where the genomes of different organisms are represented as strings of characters over the 4-letter alphabet $\{a, c, g, t\}$ of DNA bases. The size of each sequence is measured in base pairs (bp), where each base pair occupies 1 byte of storage. As of February 2008, the total size of the publicly available GenBank sequence databases has reached 85Gbp, and the size of data in the Whole Genome Shotgun (WGS) sequencing project stands at about 109Gbp [27]. Notably, the size of GenBank is doubling approximately every 18 months [2].

It has become clear that if we want to use the information from genetic databases to its full potential, we need to design more efficient techniques to support user-defined queries for this data. One of the promising directions is the preprocessing of entire collections of genomic data into indexing structures which facilitate efficient query evaluation. Since the nature of DNA strings does not allow the use of word indexes, we need to look at full-text indexes, i.e. indexes for all the different substrings of a given set of strings. Examples of such indexes are: suffix trees [17], suffix arrays [16], and string B-trees [7]. In this paper we consider suffix trees which are *crucial* for many applications on genome data (cf. [9]).

Once the suffix tree is built, we can solve many combinatorial problems on strings in optimal time. Finding all the substrings common for a set of strings in time linear in the length of the input is one example of such a problem [9]. Counting the total number of

[1] The preliminary version of this paper was presented as a short paper at CIKM 2009 conference

different substrings, also in linear time, is another example [22]. Further, suffix trees can be used to efficiently find all the locations of a pattern in a set of strings, to accelerate approximate pattern matching [25, 18], to compute matching statistics, to locate all repetitive substrings, or to extract palindromes [9].

As suffix trees are significantly larger than their input sequences and quickly outgrow the main memory, until recently, the theoretically attractive properties of suffix trees have not been fully exploited since the classical construction algorithms were only able to produce trees that fit in main memory.

Clearly, constructing suffix trees for larger inputs calls for disk-based methods. Thus far, two major bottlenecks have prevented an efficient construction of suffix trees in secondary storage: random traversals of the tree during its construction and massive random accesses to the input string.

Recently, research has been done toward overcoming the first bottleneck by reducing the number of random accesses to the tree on disk during its construction. This approach resulted in a number of efficient practical algorithms [1, 19, 23], each of which can build the suffix tree on disk for a set of eucaryotic genomes in a matter of hours (the size of the input is in the order of a few Gbp).

However, the scalability of all these proposed methods does not go beyond inputs that fit into main memory. As was mentioned in [24], the suffix-tree construction for the case when the input string is on disk can take weeks and even months due to a prohibitive number of random disk I/Os.

We note that indexing strings that do not fit in main memory, from a utility point of view, is much more important than indexing strings that do. This is because for strings fitting into main memory, the on-line in-memory search methods might be more efficient than methods based on disk-based indexes.

There are theoretical results for the external memory suffix tree construction. The suffix tree algorithm by Farach et al. [6] is theoretically optimal for the external memory computational model. The time complexity of this algorithm is equal to the complexity of sorting. Nevertheless, as discussed for example in [22], the overall intricacy of the Farach algorithm has prevented, so far, its practical implementation.

On the other hand, the currently implemented algorithms for the suffix tree construction do not scale for input strings which do not fit in main memory. Thus, how to design an efficient practical algorithm to build suffix trees for such strings remains an open problem. The challenge can be formulated as follows: can multiple random accesses to the input string be avoided during the suffix tree construction?

The main contribution of this paper is the first practical algorithm for constructing suffix trees for inputs larger than the size of the main memory. Specifically, we make the following contributions:

1. We present $B^2ST$, an efficient external-memory suffix tree construction algorithm for very large inputs.[2] The algorithm is based on partitioning the input, sorting the suffixes in each partition pair, and efficiently merging the sorted suffixes into a suffix tree. Our $B^2ST$ algorithm minimizes random access to the input string, and accesses the disk-based data structures sequentially.

[2] $B^2ST$ stands for **B**ig string, **B**ig **S**uffix **T**ree

2. We show that $B^2ST$ scales to much larger inputs than the previous algorithms. Our algorithm is able to build a disk-based suffix tree for virtually unlimited size of input strings, thus filling the ever growing gap between the increase of main memory in modern computers and the much faster increase in the size of genomic databases. For example, using our implementation of $B^2ST$ we build the suffix tree for a DNA sequence of total size of about 12GB in 26 hours on a single machine using only 2GB of main memory.
3. We show that $B^2ST$ is several times more efficient than the previously proposed algorithms *TDD* [24] and *Trellis+SB* [20] designed for input strings larger than the main memory. This is because $B^2ST$ performs sequential access to both the input string and tree being built, whereas the other algorithms cannot avoid a large enough number of random accesses to the input string.

The remainder of this paper is organized as follows: the research related to the problem of building suffix trees for inputs in excess of RAM is discussed in Section 2, the detailed description of the new algorithm is given in Section 3, and in Section 4 we give an experimental evaluation of the proposed algorithm.

## 2 Related work

The performance of the best practical algorithms for disk-based suffix tree construction degrades rapidly when the input string does not fit the main memory [1, 19, 23]. All these algorithms are designed to rely on random access to the input string and on minimizing the random access to the partially built tree on disk. Each of these methods was extended to handle the case of extra large inputs. The results of these efforts are summarized below, clearly indicating that the problem is far from being solved.

### 2.1 ST-merge and the locality of references

In [24], the Top Down Disk based suffix tree construction algorithm *TDD* was extended for the case when the input string does not fit the main memory. The authors proposed the *Suffix Tree Merge (ST-Merge)* algorithm, which works as follows. The input of size $N$ is partitioned into $K$ partitions, such that $N/K$ is smaller than the size of the main memory. A suffix tree for each partition is built using the *TDD* algorithm. After this, suffix trees from different partitions are merged into a single tree. In the merge step, the suffix-tree edges from the different partitions are grouped by their first character. Next, their *longest common prefix* (LCP) is calculated by scanning the corresponding parts of the input. The result of this calculation produces a new internal node in the growing suffix tree. Then the process continues for the sub-groups of suffixes which differ in the character next to LCP. The merge performed by *ST-merge* involves multiple random accesses to the input string.

This algorithm was expected to have better locality of references in the access to the input string than the original *TDD* algorithm. However, the experimental evaluation reported in [24] has shown that the *ST-merge* algorithm runs an infeasible amount of time for moderate input sizes. For example, the construction of the suffix tree for an input of size 20MB using 6MB of main memory (allocated for the input string buffer) took about 8 hours. The performance for larger inputs was not reported. Since the improvement over

the original *TDD* algorithm was insignificant,[3] in our comparative experiments we use *TDD* instead of *ST-merge*.

### 2.2 Trellis with string buffer

Interesting original ideas to overcome the input string bottleneck were proposed by the authors of the *Trellis* algorithm in [20] where they developed a new version of the algorithm – *Trellis with String Buffer* (*Trellis+SB*). Similar to *ST-merge*, the *Trellis* algorithm is based on a partition and merge strategy. It first builds suffix trees for partitions of the input string. Then it breaks the suffix tree of each partition into sub-trees according to the precomputed set of prefixes. Next, the sub-trees from different partitions which share the same prefix are merged together. The total size of sub-trees for each common prefix allows all such sub-trees to be merged in main memory. In the merge phase, however, the edges are compared character-by-character, incurring massive random accesses to the entire input string. This requires the entire input string to reside in main memory, otherwise the performance severely degrades.

To improve this behavior during the merge, in *Trellis+SB* some parts of the input string are kept in the main memory. The rest is read from disk when required. Since suffix-tree edges contain positions of the corresponding substring inside the input string, *Trellis+SB* replaces these positions, whenever possible, by positions in one small representative partition. This small representative part of the input is kept in memory during each merge and increases the buffer hit rate. Another technique used by *Trellis+SB* is the buffering of some initial characters for each leaf node.

The combination of these techniques allowed in practice to reduce the number of accesses to the on-disk input string by 95%. Note that the remaining 5%, for example, for an input of 10GB correspond to 500 million of random disk I/Os. The authors report that they were able to build the suffix tree for 3GB of the Human genome, using 512MB of main memory, in 11 hours on an Apple Power Mac G5 with a 2.7GHz processor and 4GB of total RAM. The performance for larger inputs was not reported.

### 2.3 DiGeST and prefix buffering

Similar results were obtained for the *DiGeST* algorithm [1], which merges suffix arrays built for input partitions, using a multi-way merge sort and organizing the output buffer in form of a suffix tree. In order to reduce the access to the input string in the merge phase, for each position in the suffix arrays of partitions a 32–character prefix was attached. This prefix served the comparison of suffixes of different partitions in the merge phase of the algorithm. The references to the input string were reduced by 98% for the DNA data used in the experiments in [1]. Even 2% of remaining random accesses significantly degraded the perfomance of *DiGeST* when the input string was kept on disk.

From the evaluation of all these methods, it follows that the suffix tree construction algorithms for inputs in secondary storage remain impractical and call for a better solution.

[3] The implementation of *ST-Merge* is not available [20].

## 3 Our Algorithm

### 3.1 Problem definition

We consider a *string* $X = x_0x_1 \dots x_{N-2}\$$ to be a sequence of $N$ symbols. The first $N-1$ symbols are over a finite alphabet $\Sigma$, $x_i \in \Sigma$ $(0 \leq i < N-1)$. The last symbol $x_{N-1}$ is unique and not in $\Sigma$ (a so-called *sentinel*).

By $S_i = X[i, N]$ we denote a *suffix* of $X$ beginning at position $i$, $0 \leq i < N$. Thus $S_0 = X$ and $S_{N-1} = \$$. Note that we can uniquely identify each suffix by its starting position.

*Prefix* $P_i$ is a substring $[0, i]$ of $X$. The *longest common prefix* $LCP_{ij}$ of two suffixes $S_i$ and $S_j$ is a substring $X[i, i+k]$ such that $X[i, i+k] = X[j, j+k]$, and $X[i, i+k+1] \neq X[j, j+k+1]$. For example, if $X = ababc$, the $LCP_{0,2} = ab$, and $|LCP_{0,2}| = 2$.

If we sort all the suffixes in lexicographical order, and record this order into an array of integers, then we obtain the *suffix array* $SA$ of $X$. $SA$ holds all integers $i$ in the range $[0, N]$, where $i$ represents $S_i$. In more practical terms, the array $SA$ stores integers sorted according to the lexicographical order of the suffixes these numbers represent. For example, for $X = ababc$, $SA = [0, 2, 1, 3, 4]$, since $S_0 = ababc$ is lexicographically smaller than $S_2 = abc$ etc. The suffix array can be augmented with the information about the longest common prefixes for each pair of suffixes represented as consecutive numbers in $SA$.

A *suffix tree* [17] is a digital tree of symbols for the suffixes of $X$, where edges are labeled with the start and end positions in $X$ of the substrings they represent. Note also that each internal node in the suffix tree represents an end of the longest common prefix for some pair of suffixes. Figure 1 shows what the suffix tree for $X$ looks like. The leaves are labeled with the start position in $X$ of corresponding suffixes, and each suffix can be found in the tree by concatenating substrings associated with edge labels. These substrings are not stored explicitly, but each substring is represented as an ordered pair of integers indexing its start and end position in $X$. The total number of nodes in the suffix tree is constrained due to two facts: there are exactly $N$ leaves, and the degree of any external node is at least 2. There are therefore at most $N-1$ internal nodes in the tree. Hence, the maximum number of nodes (and edges) is linear in $N$. The tree's total space is linear in $N$ in the case that each edge label can be stored in a constant space. Fortunately, this is the case for an implicit representation of substrings by their positions.

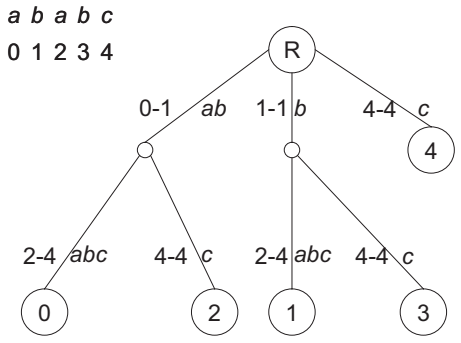


**Fig. 1.** Suffix tree for $X = ababc$. For clarity, the explicit edge labels are shown, which are represented as ordered pairs of positions in the actual suffix tree. Each suffix $S_i$ can be found by concatenating substrings of $X$ on the path from the root to the leaf node $L_i$.

In this article, we discuss the problem of constructing suffix tree $ST$ for string $X$ of size $N$. Our challenge is that the size $M$ of the main memory is smaller than the space needed to hold the entire input string; the *input-to-memory ratio* $r = N/M$ is at least 2. Therefore, neither $X$ nor $ST$ can be entirely loaded into the main memory and only some parts of them can be held in main memory buffers. The goal is to build $ST$ minimizing *random* disk I/Os.

### 3.2 Our solution

Our solution for the problem described above is based on the following ideas.

The suffix tree of $X$ can be constructed given its suffix array $SA$ augmented with an LCP information. As was proved in [6], the conversion of the suffix array into the suffix tree can be performed in a linear time. In practice, this process exhibits a good locality of references and therefore a good behavior in external memory settings. When we incrementally insert sorted suffixes into the growing suffix tree, we perform the sequential reading of the suffix array and the sequential writing of the suffix trees for consecutive lexicographic intervals, without performing random disk I/Os to both these data structures. This shows that both $SA$ and $ST$ can be kept on disk, and only their sequential parts can be loaded and manipulated in main memory.

Consequently, the first step in our algorithm for the suffix tree construction is obtaining a suffix array for string $X$, augmented with the LCP information. For this, we want to lexicographically sort all suffixes of $X$. Suffix sorting differs from conventional string sorting in that the elements to be sorted are $N$ overlapping strings, and the length of each such string is $O(N)$. This implies that a comparison-based sorting algorithm, which requires $O(N \log N)$ comparisons, may take $O(N^2 \log N)$ time. Moreover, if we treat suffixes as if they were regular strings we have an even bigger problem: when comparing a pair of suffixes we need to scan the corresponding sequences of symbols in $X$ starting at two positions along the string $X$. These positions are mostly non-consecutive. When $X$ is on disk, this translates into a prohibitive number of random disk I/Os.

The second idea we use in the design of a new algorithm is the general paradigm of the external memory two-phase multi-way merge-sort *(2PMMS)* [8]. We partition $X$ into slightly overlapping substrings (partitions) and lexicographically sort the suffixes in each partition. We can do this in main memory by using any of the best algorithms for in-memory suffix sorting. Then, we output to disk the suffix arrays for suffixes in different partitions.

A problem arises when we want to merge these suffix arrays. In the simple case of merging sorted lists of keys, the relative order of elements from any two different lists is determined by comparing these elements. However, in our case, all we have are the starting positions of suffixes from different partitions, since this is all the information we can store in suffix arrays. This does not help in determining the relative lexicographical order of suffixes, since our sorting keys are the substrings of $X$ (and not their starting positions).

A naïve approach would compare two suffixes from different partitions by randomly accessing $X$, which is on disk. This would lead to $O(N)$ *random* disk I/Os. This takes a prohibitive amount of time even for small $N$. Be reminded that the size $N$ of our input string is several times larger than the available main memory.

We now present our **B***ig tree,* **B***ig string* **S***uffix* **T***ree* construction algorithm ($B^2ST$) which minimizes the number of random accesses to the input string during the merge phase.

Note, that we never load an entire input string into main memory. In the merge step we do not compare the actual input string characters, but rather deduce the necessary information from the relative order and the LCP of any two suffixes stored in specific structures which we call *pairwise order arrays*.

This new technique presents a practical method for building suffix trees from the input strings of size several times larger than the main memory.

Algorithm $B^2ST$ proceeds in three steps: input partitioning, sorting of suffixes for each pair of partitions and merging all suffixes into a disk-resident suffix tree.

### 3.3 Step 1: Input Partitioning

Our algorithm first partitions the input string $X$ of size $N$ into $k$ partitions, such that $k = 2r$ (recall that $r = N/M$ is the input to memory ratio). Note that the sequenced genomes are already partitioned into natural partitions - the chromosomes. The size of the largest sequenced chromosome, Human chromosome I, is just 247Mbp. In general, if one of the natural partitions is too large, it is partitioned into several artificial partitions, such that each partition has length at most $p$. We append to the end of each partition, except the last one, a small "tail", namely the prefix of the next partition. The tail of the partition must never occur as a substring of this partition. It serves as a sentinel for the suffixes of the partition, and its positions are not included into the suffix array of the partition, and its positions are not included into the suffix array of the partition. This ensures that each suffix $S_i$ of a partition $X_u$ ($0 \leq i < |X_u|$, $0 \leq u < k$) will be sorted as a valid representative of a suffix $S_{up+i}$ of $X$. In practice, for real-life DNA sequences, the length of such a tail is negligibly small compared to the size of the partition itself (it never exceeded 1000 characters in our experiments with DNA databases).

Consider the example in Figure 2. It shows the partitioning of input string $X =$ *ababaaabbabbbabaabab* into four partitions. The main memory can hold up to 16 characters of the input at a time. To facilitate the example, we represent our input as a binary string ($a$ stands for 0-bit, $b$ stands for 1-bit). In this illustration we also refer to the partition numbers as $A$, $B$, $C$, and $D$ in order to distinguish them from the numbers representing character positions. Note that the tail of partition $X_B$ is substring *bbb*, which never occurs inside $X_B =$ *aabba*.

| Partition A | | | | | Partition B | | | | | Partition C | | | | | Partition D | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | a | b | a | a | a | b | b | a | b | b | b | a | b | a | a | b | a | b |
| 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |

**Fig. 2.** Input string $X =$ *ababaaabbabbbabaabab* and its four partitions. The tail of partition $B$ is substring *bbb*, which serves as a sentinel for suffixes of $X_B =$ *aabba*. The combined size of each pair of partitions with their tails must be less than the size of main memory $M$.

Note that we require the combined size of any pair of partitions $2p$ including their tails to be less than $M$.

```
Algorithm pairwiseSorting
   input: k partitions of string X
1. for (u=0; u<k-1; i++)
2.    for (v=1; v<k; v++)
3.       concatenate X_u X_v and load into RAM
4.       build suffix array with LCP SA_uv
5.       during sequential scan of SA_uv
6.          if v==k-1  //last chunk
7.             output to disk SA_v
8.          output to disk SA_u
9.          output to disk R_uv          //order array
```

**Fig. 3.** Algorithm for pairwise sorting of suffixes in all partition pairs.

### 3.4 Step 2: Suffix sorting in partition pairs

In this step we generate suffix arrays for each pair of partitions. The pseudocode for this step is shown in Figure 3. We concatenate every possible pair $u, v$ of partitions with their tails ($0 \leq u < k-1$, $u+1 \leq v < k$, $u < v$) into string $X_u X_v$. We load this input into the main memory and build the suffix array $SA_{uv}$ with attached LCP length information for each suffix. The suffixes which start in tail positions are excluded from the output suffix array, they serve only for determining the relative order of suffixes starting at the end of each partition. An LCP entry of $SA_{uv}$ is the *length* of the longest common prefix of each suffix in $SA_{uv}$ with its immediate predecessor. Figure 4 shows such an array for the pair $A$, $B$ of partitions for the same input string as in Figure 2. From each $SA_{uv}$, we extract

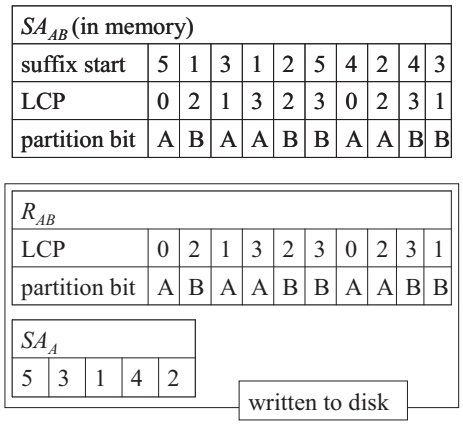


**Fig. 4.** Suffix array $SA_{AB}$ with LCP information for a pair of partitions $A$ and $B$. Two structures are extracted from $SA_{AB}$: (1) the suffix array of partition $A$ and (2) the order array $R_{AB}$ storing the relative order of suffixes in $A$ and $B$. These two structures are written to disk.

two structures: (1) the suffix array $SA_u$ for partition $X_u$ and (2) an "order array" $R_{uv}$ of size $|X_u| + |X_v|$. The *order array* $R_{uv}$ contains the $LCP$ entries of $SA_{uv}$ plus the partition

ID information. Since each $R_{uv}$ contains an information only about two partitions, we only need to use *one bit* to represent the partition ID in $R_{uv}$. Specifically, we use 0 for $u$ and 1 for $v$ ($u < v$). Figure 4 shows $SA_A$ and $R_{AB}$ extracted from $SA_{AB}$ for partition pair ($A$,$B$).

At the end of this step we have on disk $k$ suffix arrays for $k$ partitions (of total size $N$), plus $k(k-1)/2$ order arrays for each possible pair of partitions (of total size $kN$).

This is all the information we need to efficiently perform the merge. As a result of this merge we produce the suffix tree for the entire input string $X$. We are doing this without loading the entire input string into main memory. In fact, we never access $X$ anymore.

```
Algorithm initializeMerge
1. for each SA_buf_u
2.     read first m start positions
           from disk suffix array SA_u
3. for each R_buf_uv
4.     read first m/k LCP+partitionBit from R_uv

5. for each SA_buf_u
6.     insert SA_buf_u[0] into heap
```

**Fig. 5.** The pseudocode for buffer allocation as the initial step for merge.

### 3.5 Step 3: Merging

In order to merge the suffix arrays of different partitions, we use the information from the order arrays. Notably, all these arrays are accessed sequentially.

More specifically, the merge works as follows. As in the classical *2PMMS*, we have $k$ input buffers for each of the $k$ disk-based suffix arrays created in Step 2. We denote the buffer for a suffix array $SA_u$ by $SA\_BUF_u$.

In addition, we use $k(k-1)/2$ input buffers for order arrays. We denote the buffer for an order array $R_{uv}$ by $R\_BUF_{uv}$.

Finally, we have an output buffer, $ST\_BUF$, where we collect the nodes of the merged suffix tree before emptying it to disk. The total size of all the buffers matches the size of the available main memory.

We start with filling the input buffers with the elements of the corresponding arrays (See pseudocode in Figure 5). We associate a pointer with each $SA\_BUF$ and $R\_BUF$ which points to the current element of the buffer. Originally, the pointers are set to the first element of each buffer.

We start by comparing the first elements of each suffix array buffer. In order to compare two entries of, say, $SA\_BUF_u$ and $SA\_BUF_v$ ($u < v$), we consult the partition bit in buffer $R\_BUF_{uv}$ under the current pointer, as shown in pseudocode of Figure 6. If the bit is 0, we conclude that the current suffix of partition $u$ is lexicographically smaller than the current suffix of partition $v$, and vice versa.

The top element of the heap, the smallest suffix (belonging to, say, a partition $u$) migrates to the suffix-tree output buffer. The pointer for buffer $SA\_BUF_u$ is advanced by 1. The

```
Algorithm compareSuffix (S_i from partition u,
                         S_j from partition v)
1.  if (u = = v)
2.     return -1            //S_i <_lex S_j, since they are sorted
                            //in increasing order inside each partition
3.  if (u < v)
4.     if (partitionBit in R_buf_uv[current pointer] = = 0)
5.        return -1                    //S_i <_lex S_j
6.     else
7.        return 1                     //S_i >_lex S_j
8.  if (u > v)
9.     if (partitionBit in R_buf_vu[current pointer] = = 0)
10.       return 1                     //S_j <_lex S_i
11.    else
12.       return -1                    //S_j >_lex S_i
```

**Fig. 6.** Algorithm for suffix comparison which uses the pairwise suffix information from the order arrays created during pairwise suffix sorting.

pointers for all the order array buffers containing information about partition $u$ are also advanced by 1, as shown in pseudocode of Figure 7. This means we have determined the order of the current suffix of partition $u$, and we need to consider the next element both in $SA\_BUF_u$ and in all relevant order buffers. We insert into the heap the next suffix of partition $u$, and we continue in a similar way until all suffixes are merged.

```
Algorithm advancePointers (partition ID u)
1.  SA_buf_u.current_pointer++
2.  if reached the end of SA_buf_u
3.     refill SA_buf_u from disk-based SA_u
4.  for (i=0; i<u; i++)
5.     R_buf_iu.current_pointer++
6.     if reached the end of R_buf_iu
7.        refill R_buf_iu from disk-based R_iu
8.  for (i=u; i<k; i++)
9.     R_buf_ui.current_pointer++
10.    if reached the end of R_buf_ui
11.       refill R_buf_ui from disk-based R_ui
```

**Fig. 7.** Pseudocode of current pointer advancing in suffix array buffer and the order buffers.

At any point, when one of the input buffers is processed till the end, we refill it with the elements of the corresponding on-disk array. If no data remains in the on-disk array, this array is considered no longer active. When only one active $SA\_BUF$ remains, the algorithm finishes up by just adding all the remaining suffixes of this array to the (output) suffix tree.

Note, that the disk-resident suffix arrays and the order arrays are read sequentially, which would not be the case if we were consulting the input string $X$ to resolve a relative order for arbitrary suffix start positions of different partitions.

The complete pseudocode of merge is shown in Figure 8.

```
Algorithm merge
 1.  lastTransferred = null
 2.  while heap is not empty
 3.     remove smallest suffix S_i of partition u
                          from the top of the heap
 4.     rebalance heap

 5.     lcp = 0
 6.     if lastTransferred is not null
 7.        v = lastTransferred.partitionID
 8.        lcp = LCP from R_buf_uv [current_pointer]

 9.     create leaf for S_i using lcp in ST_buf
10.     advancePointers (u)

11.     lastTransferred=S_i

12.     if ST_buf is full
13.        store S_i (max suffix)
                 as a pointer to the current tree
14.        write ST_buf to disk
15.        lastTransferred = null

16.     S_j = get next suffix from SA_buf_u
17.     if S_j is not null
18.        insert S_j into heap
```

**Fig. 8.** The general pseudocode for merge.

What happens with each suffix in the output buffer is the subject of the next section.

### 3.6 Suffix Tree Output Buffer

Here we discuss how we incrementally build the suffix tree in output buffer $ST\_BUF$.

First of all, we consider each suffix to be inserted into the tree as a sequence of bits. Therefore, we build a suffix tree over a binary alphabet. Note that since in each step we add one suffix to the tree, treating the suffix as a sequence of bits does not increase the total number of leaves in the suffix tree: the tree has one leaf node and one internal node for each inserted suffix. All that changes is the length of the edge labels. The tree over the binary alphabet is illustrated in Figure 9 which shows the suffix tree for $X_B = 0001000110$ which is an equivalent binary representation of $X = ababc$ in Figure 1.

This representation of the suffix tree supports all the usual string queries. For example, in order to find occurrences of a pattern in string $X$ we can treat the pattern as a sequence of bits, and match these bits along the path from the root of the suffix tree for $X$. Also, if we are looking for the longest repeating substring ($LRS$) of $X$, and the alphabet contains characters represented by $b$ bits each, we find the internal node of the greatest depth, say

$d$, from the root, and then we calculate the $LRS$ (with respect to the original alphabet) as $LRS = \lfloor d/b \rfloor$.

The advantage of using the binary alphabet is that each suffix tree node branches exactly into two children, and we can represent each suffix tree node using constant space and store the entire tree as an array of nodes.

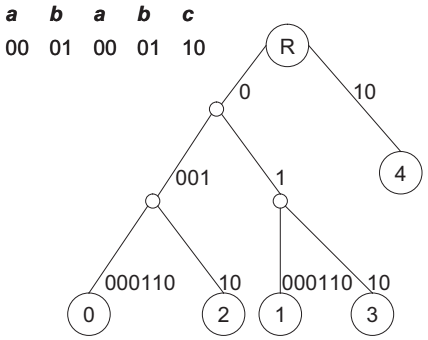


**Fig. 9.** Suffix tree over binary alphabet for $X = ababc$ with edges labeled for clarity with the corresponding bit sequences.

Thus, the suffix tree buffer $ST\_BUF$ is represented as an array of tree nodes. Each tree node is a structure containing: the positions (referring to the same array) of the left and right children of the current node, and the start position $i$ of the substring $X[i, j]$ labeling the incoming edge of this node. Note, that we do not need to explicitly store the end position $j$ of this substring, since for internal nodes this end position can be calculated from the start position of the left child and for leaf nodes it is simply $N$. However, since we are working with inputs whose size exceeds what can be stored in a 4-byte integer, we have to represent the position inside the input as a combination of the partition number and the offset inside this partition. We use 1 byte for the partition id, 2 integers (8 bytes) for the children's positions and 1 integer (4 bytes) for the substring $X[i, j]$ start position inside the partition.

$ST\_BUF$ is used to collect a growing suffix tree of lexicographically sorted suffixes. The pointer in $ST\_BUF$ points to the next available slot. When the first suffix, say $S_i$, migrates to the output buffer, we initialize the tree considering $ST\_BUF[0]$ as a root node, and advance the pointer by 1. Then, we add the first child — a leaf node for suffix $S_i$. This node $ST\_BUF[1]$ is placed into the next slot of the array. It contains start position $i$ of $S_i$ and its partition number. When the second suffix, say $S_j$, migrates into the growing suffix tree, all we need to do is to find the splitting point at a distance of $LCP_{ij}$ from the root. We use the previous leaf node $ST\_BUF[1]$ as a new internal node, and we create two leaf nodes: $ST\_BUF[2]$ for an existing leaf and $ST\_BUF[3]$ for a new leaf corresponding to a second suffix $S_j$. The left child position of the node stored in $ST\_BUF[1]$ is now 2, and the right child is 3. The start positions of the new leaves are calculated by adding $LCP_{ij}$ to $i$ and $j$ respectively. The edge to split is found on a "border path" of the current suffix tree. The *border path* is the rightmost path in the current suffix tree and it corresponds to the largest suffix inserted just before the current one.

The information about the LCP length is obtained from the corresponding order array, where it was written during the pairwise sorting step. Namely, we keep track of the partition

number of the last inserted suffix. Before adding the next suffix, we read the length of the LCP of this suffix with the previously inserted suffix (from the order array buffer).

Since we are visiting each node of the tree not more than twice, the total running time of this construction is linear in the size of the suffix tree. From the above, we have that there are two nodes for each suffix, one is a leaf and the other is internal. Therefore, $ST\_BUF$ contains suffix-tree nodes of 13 bytes each, which makes it 26 bytes of main memory per inserted suffix.

Figure 10 represents the insertion of the two first suffixes into a tree for a binary string $X_b = 0001000110$ obtained from $X = ababc$. For clarity, the array elements are aligned as tree nodes and the corresponding suffix of an incoming edge for each node is explicitly shown. Note that only suffixes which start at even positions of $X_b$ are inserted into a tree, since they represent the actual suffixes of $X$.

The construction of the tree in this example proceeds as follows. Along the LCP information in each order array we store one bit representing the next bit after —LCP— in the current suffix. The lexicographically smallest suffix $S_0$ (0001000110) is the first to be inserted into the tree. It is the left child of the root, since the first bit after $LCP$ is 0. We create a root node and set its left child value to 1 which is the next available slot in $ST\_BUF$ array. The incoming edge of the new leaf node represents the substring starting at position 0 and ending at $N$. The next suffix to be inserted is suffix $S_4$, and its $|LCP|$ with the previous suffix is 4 (in bits). We break the edge of the previously added leaf node by converting it into a new internal node at depth 4 from the root and adding two new leaves corresponding to the previous suffix $S_0$ and a new suffix $S_4$.

We continue in a similar way filling the output buffer with lexicographically ordered suffixes. When the buffer is full, it is flushed to disk, the $ST\_BUF$ array pointer is reset to position zero, and the suffix tree construction starts from the beginning.

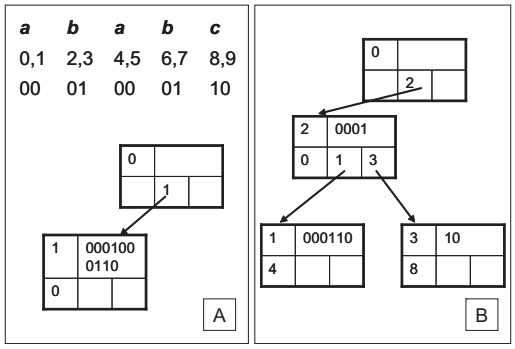


**Fig. 10.** The two first steps of building the suffix tree for $X_B = 0001000110$, which represents $X = ababc$, in output buffer $ST\_BUF$. **A**. Adding the lexicographically smallest suffix $S_0 = 0001000110$. **B**. Adding suffix $S_4 = 000110$ by splitting the edge at distance $|LCP_{0,4}| = 4$ from the root node.

### 3.7 The suffix tree on-disk layouts

Note that, since the leaf nodes do not have any child information, after a suffix tree with $L$ leaves is built in the output buffer, we can write it to disk as two separate arrays: an array

of internal nodes of size up to $L$ and an array of leaf nodes of size exactly $L$. Each entry in the array of internal nodes has a size of 13 bytes, and each element in the array of leaf nodes has a size of 5 bytes. The total size of the on-disk suffix tree for $L$ suffixes is therefore $13 + 5 = 18$ bytes per suffix. Note that this representation of a suffix tree does not harm the performance of string queries.

Before flushing the output buffer to disk, as done in [1], we add the 32-character prefix of the suffix last inserted into the tree to a collection of *dividers*. Each divider contains the prefix of the lexicographically largest suffix in the corresponding suffix tree along with a pointer to the file where the tree is stored.

Thus, at the end of the computation we have on disk a forest of suffix trees, each of which can quickly be located from a corresponding divider, and then loaded and queried in main memory. Notably, all these small trees are of equal length, which solves the problem of data skew for prefix-based partitioning of the on-disk suffix tree, as in [10, 23].

Finally, we note that the collection of dividers is small and can be kept in main memory during query execution. For example, in order to locate a pattern in $X$, we first scan the collection of dividers to find the proper tree, then we load this tree into main memory and search inside it.

### 3.8 Analysis

Since the suffix array construction and the LCP computation for each pair of partitions can be done in time linear in its length $2N/k$, and we have $k(k-1)/2$ different pair combinations, the running time of the sorting step is $O(kN)$, where $k = 2N/M$. In other words, the time is proportional to the total input size and the input-to-memory ratio, i.e. how many times our input exceeds the available main memory. The suffix arrays and the order arrays produced in this step require $O(kN)$ temporary disk space. In the merge step, the suffix tree of the size $O(N)$ is constructed in time linear in $N$, this requres, however, the complete scan of the intermediate order arrays of size $O(kN)$. Thus, the total running time of the $B^2ST$ algorithm is $O(kN)$.

## 4 Experimental Evaluation

We implemented $B^2ST$ in C and compiled with a GNU gcc compiler, version 4.1.2. All experiments were performed on a machine with an Intel Core Duo 2.66 Ghz CPU, 2GB RAM and 4MB L2 cache under Ubuntu 7.04, 32-bit Linux.

The first step in our implementation was to choose an algorithm which creates a suffix array with $LCP$ information for each pair of partitions. Our objective was to evaluate the main idea of $B^2ST$, assuming the required suffix arrays for each pair of partitions are given. For this purpose we used the modification of *DiGeST* [1] which outputs a suffix array with LCP values instead of suffix trees. *DiGeST* can build suffix arrays for very large inputs, requiring only that the input string fits into main memory. Thus, suffix arrays for 4GB of DNA are efficiently built using a main memory string buffer of 1GB assuming the DNA string is first compressed using 2 bits per character.

We first evaluated the performance of our algorithm in comparison with algorithms *TDD* and *Trellis+SB*. We obtained the source code for *Trellis+SB* and *TDD* from [29] and

[28] respectively. Since *TDD* and *Trellis+SB* implementations were not reported to handle inputs larger than 3GB, we designed a comparative experiment for 3GB of human genome DNA input.

Both *Trellis+SB* and $B^2ST$ work on compressed inputs (using two bits per DNA character). This means that the 3GB of DNA occupies 750MB of space. In order to simulate the case when the input string does not fit into main memory, we restricted the total available memory to these algorithms to 600MB. We have used the maximum of available main memory (2GB) for *TDD* which works with uncompressed inputs. The results are shown in Figure 11.

| Program | Time, hours |
|---|---|
| TDD | 125 |
| Trellis+SB | 11 |
| $B^2ST$ | 3 |

**Fig. 11.** Running times of different suffix tree construction algorithms for approximately 3GB of DNA sequence (human genome) which is larger than the total allocated main memory.

*TDD* builds the suffix tree for the above 3GB input in 125 hours. We were unable to reproduce the results of *Trellis+SB* reported in [20]. The value in Figure 11 is the result reported in [20] for similar settings on a comparable machine.

For $B^2ST$ we divided the 3GB into partitions of 1GB each and built the suffix array for partition pairs of a total size of 2GB. As already mentioned, we used for this 600 MB of main memory. We then merged the arrays using the technique described in Section 3 into suffix trees of a total size of 59GB.

The sorting of suffixes in the 3 pairs of 3 partitions took 118 minutes, while the merge took only 13 minutes. This shows that our new algorithm $B^2ST$ achieves a drastic performance improvement over the other algorithms for strings that do not fit the main memory.

This confirms that performing sequential scans as we do in $B^2ST$ pays off compared to just focusing on reducing the number of random disk I/O's to the input string, as the other algorithms do.

Next we evaluated the scalability of our algorithm. Using 1.5GB of main memory to hold input string we constructed suffix trees for 6, 8, 10 and 12GB of genomic data. These datasets were generated from combinations of sequences of eucaryotic genomes obtained from [26]. The exact description of inputs is as follows:

1. A dataset of size 6.2GB containing human, chimpanzee, and zebra fish genomes
2. A dataset of size 8.4GB containing human, chimpanzee, zebra fish, and cow genomes
3. A dataset of size 9.7GB containing human, chimpanzee, zebra fish, mouse, and chicken genomes
4. A dataset of size 11.7GB containing human, chimpanzee, zebra fish, cow, mouse, and chicken genomes

The performance results are shown in Figure 12. The size of each partition is 2GB. The partition pair of size 4GB was compressed into 2 bits per character string, and processed into the suffix array with LCP. For our largest input, 12GB, we had 6 partitions and 15 partition pairs. The time taken to build the suffix arrays of these 15 pairs was 25 hours

and produced an intermediate on-disk output of size 234GB. Despite this, the merge phase completed in only 59 minutes, scanning all this on-disk data in sequential manner and produced 2514 suffix tree files of total 215GB.

| Input size, GB | Number of partitions | Number of partition pairs | Pairwise sorting, min | Merge, min | Total time, min |
|---|---|---|---|---|---|
| 6 | 3 | 3 | 441 | 27 | 468 |
| 8 | 4 | 6 | 730 | 34 | 764 |
| 10 | 5 | 10 | 1100 | 42 | 1142 |
| 12 | 6 | 15 | 1462 | 59 | 1521 |

**Fig. 12.** Running time (min) of $B^2ST$ for different sets (approximately 6, 8, 10 and 12GB) of genomic DNA using only 2GB of main memory.

This example shows that we need a large temporary disk space for scaling up the $B^2ST$ algorithm. Specifically, we need $D = k^2p = kN$ disk space to store the order arrays for all partition pairs. Since the number of partitions is $k = N/M$, from $D = N^2/M$ we can determine the size of the largest input that we can process with $M$ bytes of internal memory and $D$ bytes of disk space. If we substitute the common values for modern computers, $D = 10^{12}$ (1TB), and $M = 4 \times 10^9$ (4GB), then we can build suffix trees using such a machine for up to 60GB of input. Note, however, that the construction of suffix trees even for 10GB of input was never achieved and reported before.

As for the execution time, it is clear that the construction of suffix arrays for a pair of partitions can be done in parallel, since each such sorting is independent of the others. The scanning of $O(kN)$ intermediate disk structures in the merge step is very efficient due to the sequential reading. So, by using our $B^2ST$ algorithm, indexing a large amount of DNA data with suffix trees becomes a feasible routine task.

### 4.1 Concluding remarks

In this paper we described a new approach to build suffix trees for very large inputs. The $B^2ST$ algorithm is designed to store all its inputs, outputs and intermediate data structures on disk, making it a truly external memory algorithm. The algorithm uses an external memory merge-sort paradigm and allows to scale up the building of suffix trees on disk for inputs, which was practically impossible before.

## References


1. M. Barsky, U. Stege, A. Thomo, and C. Upton. A new method for indexing genomes using on-disk suffix trees.
   *Proceedings of CIKM 2008*: 649–658, 2008.
2. D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and D. Wheeler. GenBank.
   *Nucleic Acids Research*, 34: 2006.
3. W.I. Chang, E.L. Lawler. Sublinear Approximate String Matching and Biological Applications.
   *Algorithmica*, 12(4/5): 327–344, 1994.

4. A. Crauser, and P. Ferragina. Theoretical and experimental study on the construction of suffix arrays in external memory.
*Algorithmica*, 32(1): 1–35, 2002.
5. 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.
6. 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.
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. H. Garcia-Molina, J. D. Ullman, J. D. Widom. Database System Implementation.
Prentice-Hall Inc., 1999.
9. D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.
Cambridge University Press, 1997.
10. E. Hunt, M.P. Atkinson, R.W. Irving. A database index to large biological sequences.
*The VLDB Jornal*, 7(3): 139–148, 2001.
11. M.V. Katti,P.K. Ranjekar, and V.S. Gupta. Differential distribution of simple sequence repeats in eukaryotic genome sesquences.
*Molecular Biology and Evolution*, 18: 1161–1167, 2001.
12. J. Kececioglu, and J. Ju. Separating repeats in DNA sequence assembly.
*Proceedings of RECOMB 2001*: 176–183, 2001
13. D. Knuth. The Art of Computer Programming, Volume 3: Sorting and Searching.
Addison-Wesley, 1998.
14. S. Kurtz, A. Phillippy, A.L. Delcher, et al. Versatile and open software for comparing large genomes.
*Genome Biol*, 5(R12): 2004.
15. S. Kurtz, and C. Schleiermacher. REPuter: fast computation of maximal repeats in complete genomes.
*Bioinformatics*, 15: 426–427, 1999.
16. U. Manber, and E. Myers. Suffix Arrays: A New Method for On-Line String Searches.
*SIAM Journal of Computing*, 22(5): 935–948, 1993.
17. E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm.
*Journal of the ACM*, 23(2): 262–272, 1976.
18. G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching.
Technical Report TR/DCC-98-14 of the Department of Computer Science, Univ. of Chile, 1998.
19. B. Phoophakdee and M. J. Zaki. Genome-scale Disk-based Suffix Tree Indexing.
*ACM SIGMOD International Conference on Management of Data*, 2007.
20. B. Phoophakdee and M. J. Zaki Trellis+: An Effective Approach for Indexing Massive Sequence.
*Pacific Symposium on Biocomputing*, 2008.
21. A. Siepal et al. Evolutionarily conserved elements in vertebrate, insect, worm, and yeast genomes.
*Genome Research*, 15 : 1034–1050, 2005.
22. W. Smyth. Computing Patterns in Strings.
Addison-Wesley, 2003.
23. S. Tata, R.A. Hankins, J.M. Patel. Practical suffix tree construction.
*Proceedings of 30th VLDB conference*, 36–47, 2004
24. Y. Tian, S. Tata, R. Hankins, J. Patel. Practical methods for constructing suffix trees.
*The VLDB Journal*, 14(3) : 281–299, 2005.
25. J. Yang, W. Wang, Y. Xia, P. S. Yu. Accelerating Approximate Subsequence Search on Large Protein Sequence Databases.
*CSB, Proceedings of the IEEE conference on Bioinformatics*: 207, 2002.
26. USCS Genome Browser:
*hgdownload.cse.ucsc.edu/downloads.html*

27. NCBI Genebank overview:
*http://www.ncbi.nlm.nih.gov/Genbank/*
28. Source code for TDD:
*www.eecs.umich.edutddownload.html*
29. Source code for TRELLIS+SB:
*www.cs.rpi.edu/∼zaki/software/trellis*