A survey of practical algorithms for suffix tree construction in external memory

M. Barsky*,†, U. Stege and A. Thomo

University of Victoria, PO Box 3055, STN CSC Victoria, BC, V8W 3P6, Canada

SUMMARY

The construction of suffix trees in secondary storage was considered impractical due to its excessive I/O cost. Algorithms developed in the last decade show that a suffix tree can efficiently be built in secondary storage for inputs which fit the main memory. In this paper, we analyze the details of algorithmic approaches to the external memory suffix tree construction and compare the performance and scalability of existing state-of-the-art software based on these algorithms.

KEY WORDS: *suffix tree*; external memory algorithms; string index

1. Introduction

Suffix trees [27] are digital trees which index all the distinct non-empty substrings of a given set of strings. An early, implicit form of suffix trees can be found in Morrison's [28] Patricia tree†. But it was Weiner [42] who initially proposed to use a suffix tree as an explicit index.

Once the suffix tree for a set of strings is built, we can solve multiple combinatorial problems on strings in optimal time, that is in time linear in the length of the input. Finding common patterns, each pattern being a substring of every string in the input set, is one example of such a problem [17]. Counting the total number of different substrings with the same linear time complexity is another example [35]. Suffix trees can be used to find all the locations of a pattern in a set of strings, to compute matching statistics, to locate all repetitive substrings, or to extract palindromes [17].

*Correspondence to: Marina Barsky, PO Box 3055, STN CSC Victoria, BC, Canada, V8W 3P6
†E-mail: mgbarsky@cs.uvic.ca
†PATRICIA stands for **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric

Such marvelous facilities do not come without a price: the suffix tree occupies at least 10 times more space than the input it is built upon. For example, when we build the suffix tree for an input of size 1GB, we will require at least 10GB of space. 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 [43]. Notably, the size of GenBank is doubling approximately every 18 months [5]. If we aim to build the suffix tree for the entire database of publicly available sequenced DNA, the space required for such a tree (not less than 850 GB) is too big for the main memory of the modern computer. To construct such a tree, we can use larger and cheaper disk space instead[‡].

In order to use this larger disk space we need to design an external memory (EM) algorithm for the construction of the suffix tree. EM algorithms differ from the algorithms for main memory. The access to data on a disk is 10^5-10^6 times slower than the access to data in main memory [41]. In order to compensate for these speed differences in the design of EM algorithms, the external memory computational model, or **d**isk **a**ccess **m**odel (DAM), was proposed [40]. DAM represents the computer memory in form of two layers with different access characteristics: the fast main memory of a limited size M, and a slow and arbitrarily large secondary storage memory (disk). In addition, for disks, it takes about as long to fetch a consecutive block of data as it does to fetch a single byte. That is why in the DAM computational model the asymptotic performance is evaluated as the total number of block transfers between a disk and main memory.

Although the DAM computational model is a workable approximation, it does not always accurately predict the performance of EM algorithms. This is because it does not take into account the following important disk access property. The cost of a random disk access is the sum of seek time, rotational delay and transfer time. The first two dominate this cost in the average case, and as such, are the bottleneck of a random disk access. However, if the disk head is positioned exactly over the piece of data we want, then there is no seek time and rotational delay component, but only transfer time. Hence if we access data sequentially in disk, then we only pay seek time and rotational delay for locating the first block of the data, but not for the subsequent blocks. The difference in cost between sequential and random access becomes even more prominent if we also consider read-ahead-buffering optimizations which are common in current disks and operating systems [9]. Thus, the number of random disk accesses is an important measure to predict the efficiency of EM algorithms.

Before describing the strategies of EM algorithms for the suffix tree construction, let us take a closer look at the suffix tree data structure and its computer representation.

1.1. The suffix tree data structure

First, we equip ourselves with some useful definitions.

[‡]Note that we focus in our discussion on the traditional rotating disks. We believe that research on the use of SSD disks, which have a different access behavior, is certainly a promising future direction, but to the best of our knowledge, SSDs have not yet been explored as a memory extension for the suffix tree construction.

We consider a *string* $X = x_0x_1 \ldots x_{N-1}$ to be a sequence of N symbols over an alphabet Σ. We attach to the end of X one more symbol, \$, which is unique and not in Σ (a so-called *sentinel*).

By $S_i = X[i, N]$ we denote a *suffix* of X beginning at position i, $0 \leq i \leq N$. Thus $S_0 = X$ and $S_N =$ \$. 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 $X = ababc$, then $LCP_{0,2} = ab$, and $|LCP_{0,2}| = 2$.

If we sort all the suffixes of string X in lexicographical order and record this order into an array SA of integers, then we obtain the suffix array of X. SA holds all integers i in the range $[0, N]$, where i represents S_i. In more practical terms, the array SA is an array of positions sorted according to the lexicographic order of the suffixes. Note that the suffixes themselves are not stored in this array but are rather represented by their start positions. For example, for $X = ababc\$$ $SA = [5, 0, 2, 1, 3, 4]$. 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 *trie* is a type of digital search tree [24]. In a trie, each edge represents a character from the alphabet Σ. The maximum number of children for each trie node is $|\Sigma|$, and sibling edges must represent distinct symbols. A *suffix trie* is a trie for all the suffixes of X. As an example, the suffix trie for $X = ababc$ is shown in Figure 1 [Left]. Beginning at the root node, each of the suffixes of X can be found in the trie: starting with *ababc*, *babc*, *abc*, *bc* and finishing with a *c*. Because of this organization, the occurrence of any query substring of X can be found by starting at the root and following matches down the trie edges until the query is exhausted. In the worst case, the total number of nodes in the trie is quadratic in N. This situation arises, for example, if all the paths in the trie are disjoint, as for the input string *abcde*.

The number of edges in the suffix trie can be reduced by collapsing paths containing unary nodes into a single edge. This process yields the structure called *suffix tree*. Figure 1 [Right] shows what the suffix trie for X looks like when converted to a suffix tree. The tree still has the same general shape, just far fewer nodes. 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. In practice, these substrings are not stored explicitly, but they are 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: (1) there are exactly N leaves and (2) 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.

More formally, a *suffix tree* 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.

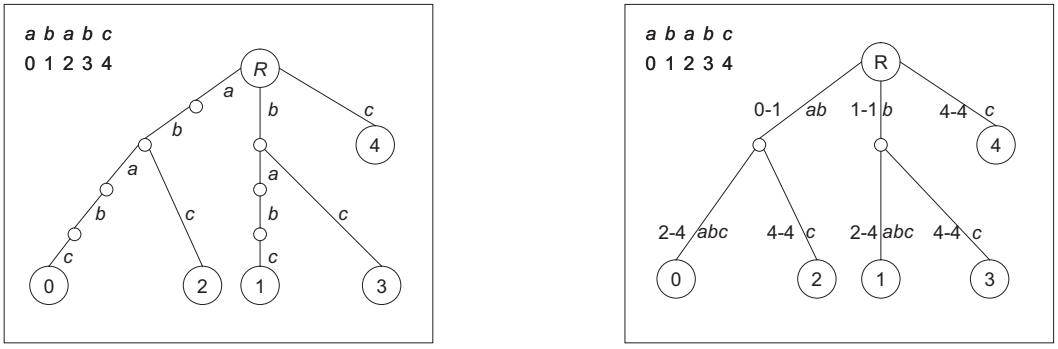

Figure 1. **[Left]** The suffix *trie* for $X = ababc$. Since c occurs only at the end of X, it can serve as a unique sentinel symbol. Note that each suffix of X can be found in the trie by concatenating character labels on the path from the root to the corresponding leaf node. **[Right]** The suffix 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.

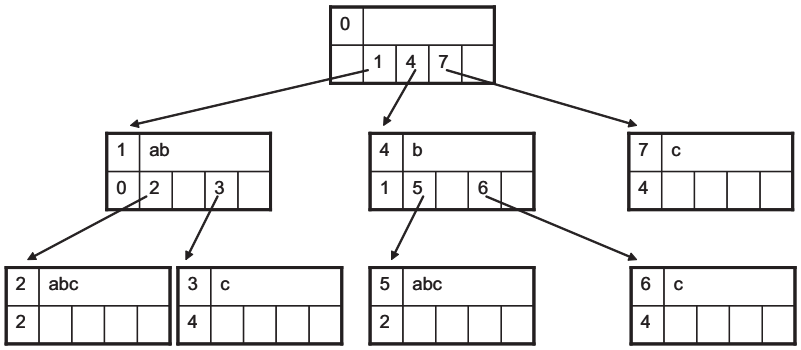

Figure 2. An array representation of the suffix tree for $X = ababc$. Each node contains an array of 4 child pointers. Note that not all the cells of this array are in use. The sequences in the nodes are the labels of the incoming edges. They are shown for clarity only and are not stored explicitly.

1.2. Suffix tree storage optimizations

We discuss next the problem of suffix tree representation in memory in order to estimate the disk space requirements for the suffix tree.

It is common to represent the node of a suffix tree together with the information about an incoming edge label. Each node, therefore, contains two integers representing the start and end positions of the corresponding substring of X. In fact, it is enough to store only the start position of this substring as the length of it can be deduced from the start position of the child node or is simply N if current node is a leaf. In a straightforward implementation, each node

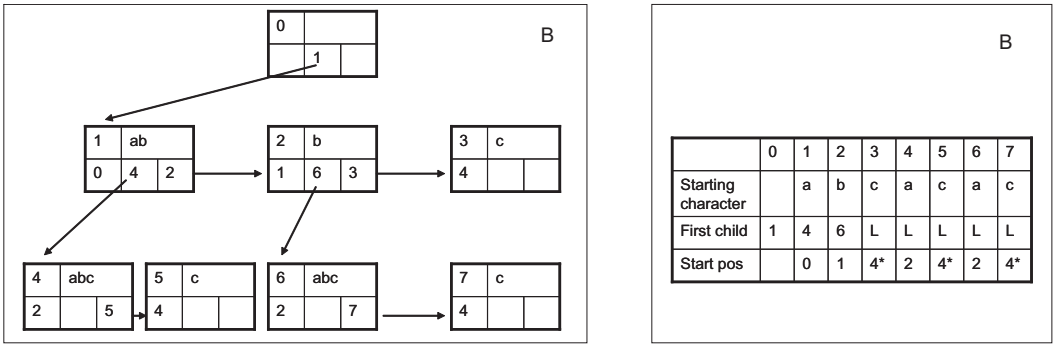

Figure 3. **[A].** Left-child right-sibling representation of the suffix tree for $X = ababc$. Each node contains 1 pointer to its first child and 1 pointer to the next sibling. **[B].** Giegerich et al.'s representation of the suffix tree, where all siblings are represented as consecutive elements in the array of nodes. The special symbol $\star$ indicates the bit representing the last sibling. Each node contains only a pointer to the first child and the start position of the incoming edge-label.

has pointers to all its child nodes. These child pointers can be represented as an array, as a linked list or as a hash table [17].

If the size of Σ is small, the child node pointers can be represented in form of an array of size $|\Sigma|$. Each i^{th} entry in this array represents the child node whose incoming label starts with the i^{th} character in a ranked alphabet. This is very useful for tree traversals, since the corresponding child can be located in constant time. Let us first consider the tree space for the inputs where N is less than the largest 4 byte integer, i.e. $\log N < 32$. In this case, each node structure consists of $|\Sigma|$ integers for child node pointers plus one integer to represent the start position of the edge-label substring. Since there are at most $2N$ nodes in the tree, the total space required is $2N(|\Sigma| + 1)$ integers, which, for example, for $|\Sigma| = 4$ (DNA alphabet) yields $40N$ bytes of storage per N bytes of input. Such representation is depicted in Figure 2.

For larger alphabets, an array representation of children is impractical and can be replaced by a linked list representation [17]. However, this requires an additional $log|\Sigma|$ search time spent at each internal node during the tree traversal, in order to locate a corresponding child. In addition, since the position of a child in a list does not reflect the first symbol of its incoming edge label, we may need to store an additional byte representing this first character.

Another possibility is to represent child pointers as a hash table [17]. This preserves a constant-time access to each child node and is more space-efficient than the array representation.

The linked-list based representation known as a "left-child right-sibling" was proposed by McCreight in [27]. In this implementation, the suffix tree is represented as a set of node structures, each consisting of the start position of the substring labeling the incoming edge, together with two pointers – one pointing to the node's first child and the other one to its next sibling. Recall that the end position of the edge-label substring is not stored explicitly, since for an internal node it can be deduced from the start position of its first child, and for a leaf node

this end position is simply N. This representation of the node's children is of type linked list, with all its space advantages and search drawbacks. The McCreight suffix tree representation is illustrated in Figure 3 [A]. Each suffix tree node consists of 3 integers, and since there are up to $2N$ nodes in the tree, the size of such a tree is at most $24N$. Again, for better traversal efficiency, we may store the first symbol along each edge label. Then the total size of a suffix tree will be at most $25N$ bytes for N bytes of input.

An even more space efficient storage scheme was proposed by Giegerich et al. [16]. In this optimization, the pointers to sibling nodes are not stored, but the sibling nodes are placed consecutively in memory. The last sibling is marked by a special bit. Now, each node stores only the start position of a corresponding edge-label plus the pointer to its leftmost child. As before, for efficiency of the traversal, each node may store an additional byte representing the start symbol of its edge label. The size of such a tree node is 9 bytes. For a maximum of $2N$ nodes this yields a maximum of $18N$ bytes of storage. Giegerich et al.'s [16] representation is depicted in Figure 3 [B].

An additional possibility to optimize the storage of the suffix tree is to consider each suffix as a sequence of bits. The problem of *renaming*, which is a generic reduction from strings over an unbounded alphabet to binary strings, was studied in [11]. It was shown that such a reduction can be done in linear time. Note that a string over any alphabet Σ can always be reduced to the binary alphabet by representing each character as a sequence of $b = log|\Sigma|$ bits and then concatenating these binary sequences.

For a binary alphabet, any internal node in the suffix tree has exactly two children. This is because such a node cannot have more than two children, but also cannot have less than two for it to be a suffix tree internal node. This allows using two child pointers only (per node) and representing the entire suffix tree as an array of the constant-sized nodes. If the entire input string is considered as a sequence of bits, only the *valid* suffixes are added to the tree. These are the suffixes starting at positions i such that $i \bmod b = 0$, where b is the number of bits used to represent each character of Σ. As such, we have the same number of tree nodes as before: the tree has one leaf node and one internal node per inserted suffix. Figure 4 shows the equivalent suffix trees over the original and the binary alphabets for input string $X = ababc$. Each node has exactly two child pointers plus one integer representing start position of incoming edge-label. Since there are exactly $2N$ nodes in such a tree, the total size is $24N$ bytes. Note that this is independent of the size of the alphabet.

This binary representation of the suffix tree supports many common 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 starting at the root. 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. Then we calculate the LRS (with respect to the original alphabet) as $LRS = \lfloor d/b \rfloor$.

Note that in all representations the leaf nodes do not contain child pointers, thus at the end of the construction we can output the leaf nodes in a separate array. Each element in the array of leaf nodes stores only the start position of the corresponding substring since the end position is implied to be N. In this case, the array representation occupies $24N$ bytes (for $\Sigma = 4$), the McCreight suffix tree occupies $20N$ bytes, Giegerich et al.'s representation occupies $12N$ bytes and the suffix tree for the binary alphabet occupies $16N$ bytes. These representations are in

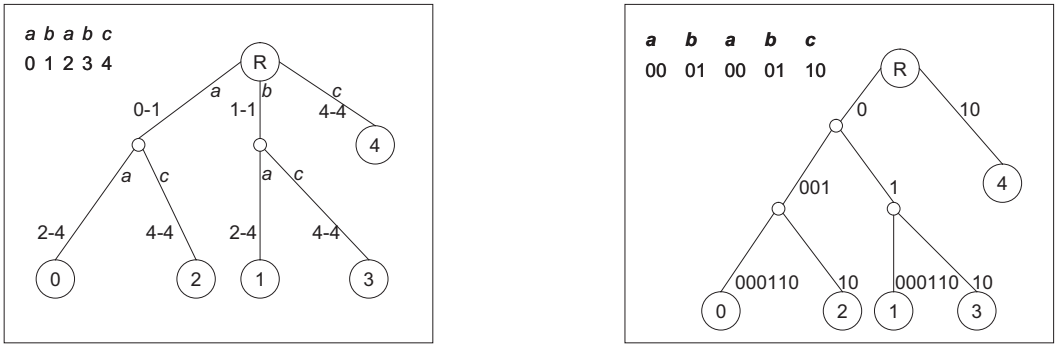

Figure 4. **[Left]** Suffix tree for $X = ababc$ given for comparison. **[Right]** Suffix tree for the same input string where each suffix is converted to a sequence of bits. Each character is encoded using 2 bits.

general[§] not well suited for the use during the process of tree construction when we update the tree nodes, but they can be used when outputting the complete tree to disk.

This short survey of storage requirements clearly demonstrates the fact that the suffix tree is very space-demanding, even if we are using an "unlimited" space of disks. For example, for an input of 2GB, the tree occupies at least 24GB of disk space. Further, for inputs exceeding in size the largest 4-byte integer, the start positions and the child pointers need more than 4 bytes for their representation, namely $\log N$ bits for each number. In practice, for the inputs of a size in the tens of gigabytes the tree can easily reach $50N$ bytes. This is important to remember while designing algorithms for efficient traversals of such large trees (see section 2.6).

Until 2007, the data structure by Giegerich et al. [16] was known as the most space efficient representation. Then Sadakane [34] fully developed the compressed suffix tree and its balanced parenthesis representation. More about compressed suffix trees can be found in recent papers [13, 33]. The compressed representation allows to store the entire suffix tree in only $5N$ bits. An example of the parenthesis representation of the suffix tree nodes for string $X = ababc$ is shown in Figure 5. The parentheses describe the tree topology. In order to store the information about the start position and the depth of each tree node, a special array and its unary encoding are used to bring the total memory requirements for the tree to $5N$ bits [34]. The compressed suffix tree supports all regular suffix tree queries with a poly-log slowdown [34]. The algorithm for the compressed suffix tree construction was implemented (see [38]) and is available for indexing genomic sequences [39]. The research on compressed suffix trees aims to compress the input string and the output tree into a smaller self-indexing structure which can fit into main

[§]The exception is the Top Down Disk Based suffix tree construction (TDD) [36], where the nodes are created from the top and at each step it is known how many children each node contains at the end of the computation. Thus, leaf nodes occupy only four bytes during the construction itself.

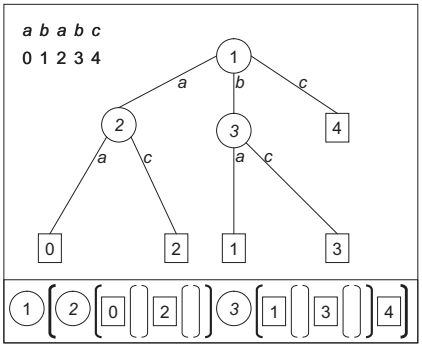

Figure 5. The parenthesis tree representation is a main high-level idea for the suffix tree compression [34].

memory. Hence, we do not consider the construction of compressed suffix trees as a part of this survey. This brief outline of the compressed suffix tree representation is given here only to show that the entire fully-functional suffix tree can be stored using much less space than previously believed. We underline that the following discussion is about the classical (non-compressed) suffix trees which are built using secondary storage.

The remainder of the paper is organized as follow. We introduce in Section 2 recent practical methods for EM suffix tree construction and evaluate their performance and scalability using the number of sequential passes over disk data, the number of random disk I/Os and the in-memory running time. Then, in Section 3 we point out the still unsolved challenges in construction of suffix trees in secondary storage. Finally, in Section 4 we outline theoretical results which may serve as a basis of further practical research.

2. Practical methods for the suffix tree construction in external memory

In this section, we present the main ideas which gave birth to the state-of-the-art software for suffix tree construction in secondary storage. These methods are steps toward a completely scalable suffix tree construction for inputs of any kind and size. When this problem is solved, a wide range of queries on massive string data will be possible to execute in optimal time.

The suffix tree for the input string X of length N can be built in time $O(N)$. Linear-time algorithms were developed in [42, 27, 37]. In [15] it was shown that all three of them are based on similar algorithmic ideas. It might be tempting to use these asymptotically optimal algorithms for an external memory implementation. However, looking closely at these algorithms, we observe that they assume that random access to the input string and to the tree takes constant time. Unfortunately, in practice, when some of these data structures outgrow the main memory and are accessed directly on disk, the access time to disk-based arrays varies significantly depending on the relative location of the data on disk. The total number of random disk accesses for these linear-time algorithms is, in fact, $O(N)$. This is extremely inefficient

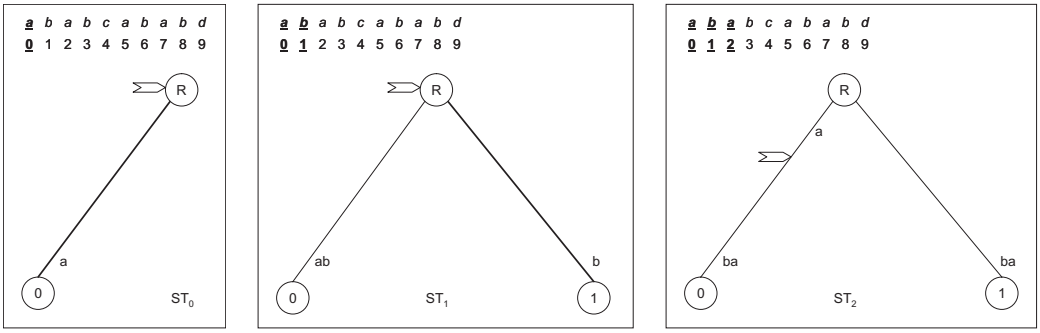

Figure 6. The three first steps of the Ukkonen algorithm. An arrow indicates the active point at the end of each iteration. Note that the extension of the edges ending at leaf nodes with the next character is performed implicitly: the edge length is just extended by 1.

and causes the so-called disk thrashing problem, which let the authors in [29] conclude that the suffix tree in secondary storage is inviable.

The random access behavior of these algorithms in external memory settings can be improved, as was shown in [4] for Ukkonen's algorithm [37]. We describe next the Ukkonen algorithm and show how it was extended for external memory.

2.1. The Ukkonen algorithm and its on-disk version

For a given string X, Ukkonen's algorithm starts with the empty tree (that is, a tree consisting just of a root node) and then progressively builds an intermediate suffix tree ST_i for each prefix $X[0,i]$, $0 \leq i < N$. In order to convert a suffix tree ST_{i-1} into ST_i, each suffix of ST_{i-1} is extended with the next character x_i. We do this by visiting each suffix in order, starting with the longest suffix and ending with the shortest one (empty string). The suffixes inserted into ST_{i-1} may end in three types of nodes: leaf nodes, internal nodes or in the middle of an edge (at a so-called *implicit* internal node). Note that if a suffix of ST_{i-1} ends in a leaf node, we do not need to extend it with the next character. Instead, we consider each leaf node as an open node: at each step of the algorithm every leaf node runs till the end of the current prefix, meaning the end position on each leaf node will eventually become N. Consider the example in Figure 6. It shows the three first iterations of the suffix tree construction for $X = ababcababd$. In the second iteration, we implicitly extend the a-child of a root node with b, and we add a new edge for b from the root (extending an empty suffix).

Thus, in each iteration, we need to update only suffixes of ST_{i-1} which end at explicit or implicit internal nodes of ST_{i-1}. We find the end of the longest among such suffixes at the *active point*. The active point is the (explicit or implicit) internal node where the previous iteration ended. If the node at the active point already has a child starting with x_i, the active point advances one position down the corresponding edge. This means that all the suffixes of ST_i already exist in ST_{i-1} as the prefixes of some other suffixes. In case that there is no

outgoing edge starting with the new character, we add a new leaf node as a child of our explicit or implicit internal node (active point). Here an implicit internal node becomes explicit. In order to move to the extension of the next suffix, which is shorter by one character, we follow the chain of *suffix links*. A suffix link is a directed edge from each internal node of the suffix tree (source) to some other internal node whose incoming path is one (the first) character shorter than the incoming path of the source node. The suffix links are added when the sequence of internal nodes is created during edge splits.

To illustrate, consider the last iteration of the Ukkonen algorithm – extending an intermediate tree for for $X = ababcababd$ with the last character d. We extend all the suffixes of ST_8 (Figure 7 [A]) with this last character. The active point is originally two characters below the node labeled by $\star$ in Figure 7 [A], and the implicit internal node is indicated by a black triangle. The active point is converted to an explicit internal node with two children: one of them is the existing leaf with incoming edge label *cababd* and the other one is a new leaf for suffix S_5 (Figure 7 [B]). Then, we follow the suffix link from the $\star$-node to the $\star\star$-node, and we add a new leaf by splitting an implicit node two characters below the $\star\star$-node. This results in the tree of Figure 7 [C] with a leaf for suffix S_6. Next, the suffix link from the $\star\star$-node leads us to the root node, and two characters along the corresponding edge we find the $\star$-node and add to it a new edge starting with d and leading to a leaf node for suffix S_7 (Figure 7 [D]). We continue in a similar manner and add the corresponding child starting with d both to the $\star\star$-node (Figure 7 [E]) and to the root (Figure 7 [F]). This illustrates how suffix links help to find all the insertion points for the new leaf nodes. There is a constant number of steps per leaf creation, therefore the total amortized running time of the Ukkonen algorithm is $O(N)$.

The pseudocode in Figure 8 shows the procedure *update* for converting ST_{i-1} into ST_i [30]. Each call of *next_smaller_suffix()* finds the next suffix by following a suffix link.

If we look at Figure's 8 pseudocode from the disk access point of view, we see that locating the next suffix requires a random tree traversal, one per leaf created. Hence, when the tree ST_{i-1} is to be stored on disk, a node access requires an entire random disk I/O. This access time depends on the disk place of the next access point. Moreover, since the edges of the tree are not labeled with actual characters, it is important that we access randomly the input string in order to compare the test_char with the characters of X encoded as positions in the suffix tree edges. Unfortunately, this leads to a very impractical performance, since the algorithm spends all its time moving the disk head from one random disk location to another.

In [4], Bedathur and Haritsa studied the patterns of node accesses during the suffix tree construction based on Ukkonen's algorithm. They found that the higher tree nodes are accessed much more frequently than the deeper ones. This gave rise to the buffer management method known as *TOP-Q*. In this on-disk version of Ukkonen's algorithm, the nodes which are accessed often, have a priority of staying in the memory buffer, and the other nodes are eventually read from disk. This significantly improves the hit rate for accessed nodes when compared to rather straightforward implementations. However, in practical terms, in order to build the suffix tree for the sequence of the Human chromosome I (approximately 247 MB), the *TOP-Q* runs for

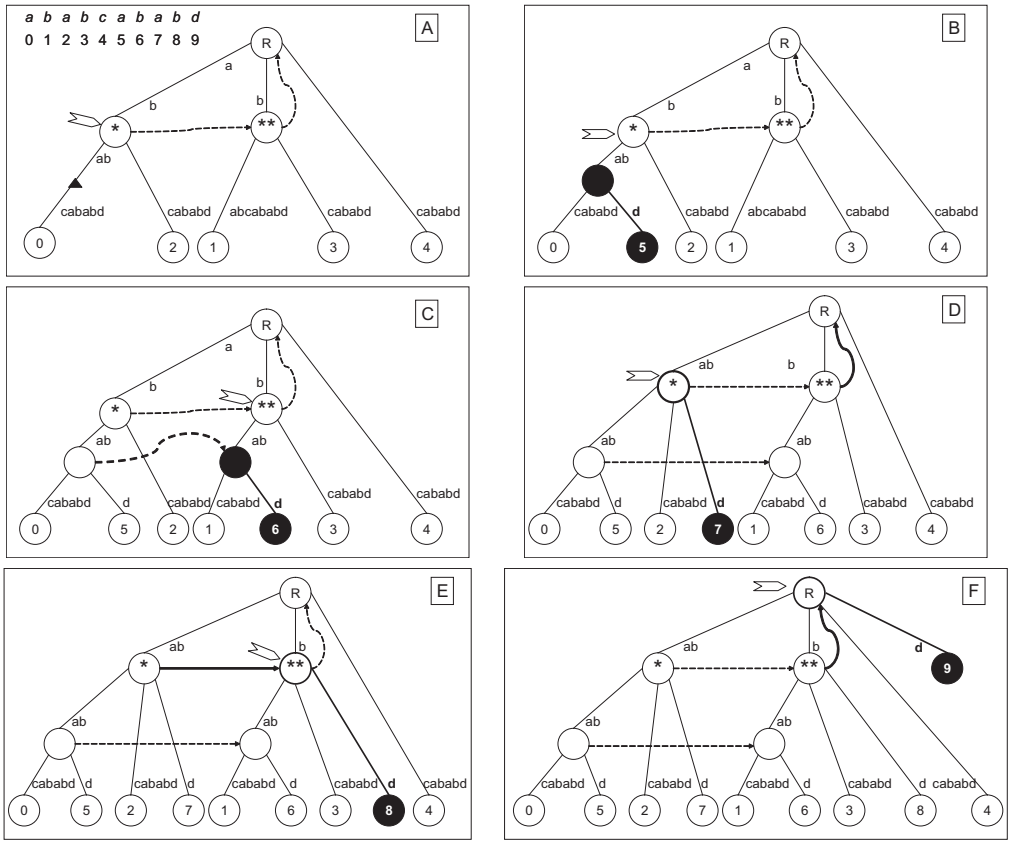

Figure 7. The last steps of the Ukkonen algorithm applied to $X = ababcababd$. In this cascade of leaf additions the ST_8 is updated to ST_9. The place for the next insertion is found following the suffix links (dotted arrows).

96 hours, as was recently evaluated using a modern machine¶[36], and can not be considered a practical method for indexing large inputs.

Next we describe a brute-force approach for the suffix tree construction, which runs in $O(N^2)$ time in the worst case. Amazingly enough, several fast practical methods for external memory were developed using this approach, due to the much better locality of tree accesses.

¶We refer to the average machine currently available (Pentium 4 with 2.8 GHz clock speed and 2 GB of main memory) as the modern machine.

```
Ukkonen's algorithm
active_point=root
for i from 0 to N
        Update ( Prefix [0,i] )

Update (Prefix [0,i] )
        curr_suffix_end = active_point
        test_char = X [i]
        done = false
        while not done
                if curr_suffix_end is located at explicit node
                        if the node has no descendant starting with test_char
                                create new leaf
                        else
                                advance active_point down the corresponding edge
                                done = true
                else
                        if the implicit node's next char is not equal test_char
                                create explicit node
                                create new leaf
                        else
                                advance active_point down the corresponding edge
                                done = true
                if curr_suffix_end is located at root node
                        active_point=root
                        done = true
                else
                        curr_suffix_end = next_smaller_suffix()  //follow the suffix link
        active_point = curr_suffix_end
```

Figure 8. Pseudocode of Ukkonen's algorithm for the suffix-tree construction.

2.2. The brute-force approach and the Hunt algorithm

An intuitive method of constructing the suffix tree ST is the following: for a given string X we start with a tree consisting of only a root node. We then successively add paths corresponding to each suffix of X from the longest to the shortest. This results in the algorithm depicted in Figure 9 [Top].

Here, ST_{i-1} represents the suffix tree after the insertion of all suffixes $S_0, \ldots S_{i-1}$. The *Update* operation inserts a path corresponding to the next suffix S_i yielding ST_i. In order to insert suffix S_i into the tree we first locate some implicit or explicit node corresponding to the longest common prefix of S_i with some other suffix S_j. To locate this node, we perform $|LCP_{ij}|$ character comparisons. After this, if the path for LCP_{ij} ends in an implicit internal node, it is transformed into an explicit internal node. In any case, we add to this internal node a new leaf corresponding to suffix S_i. Once the end of the LCP_{ij} is found, we add a new child in constant time. Finding the end of LCP_{ij} in the tree defines the overall time complexity of

Brute-force algorithm

```
for i from 0 to N
        Update ( Suffix [i,N] )

Update ( Suffix [i,N] )
        find LCP of Suffix [i,N] matching characters from the root
        if LCP ends in explicit node
                add child leaf labeled by X[i+LCP+1,N]
        else
                create explicit node at depth LCP from the root
                add child leaf labeled by X[i+LCP+1,N]
```

Hunt et al.'s algorithm

```
for each prefix PR of length prefix_len
        for i from 0 to N
                Update ( Suffix [i,N], PR )
        write sub-tree for prefix PR to disk

Update ( Suffix [i,N], PR )
        if X[i,i+prefix_len] equals PR
                find LCP of Suffix [i,N] matching characters from the root of the sub-tree
                if LCP ends in explicit node
                        add child leaf labeled by X[i+LCP+1,N]
                else
                        create explicit node at depth LCP from the root
                        add child leaf labeled by X[i+LCP+1,N]
```

Figure 9. **[Bottom]**. The pseudocode of Hunt et al.'s algorithm [18] for the suffix-tree construction based on the brute-force algorithm shown at the **[Top]**.

the algorithm. The end of a LCP can be found in one step in the best case but in N steps in the worst case for each of N inserted suffixes. This can, in the worst case, lead to $O(N^2)$ total character comparisons. However, Apostolico and Szpankowski have shown in [2] that on average the brute-force construction requires $O(N \log N)$ time. Their analysis was based on the assumption that the symbols of X are independent and randomly selected from an alphabet according to a given probability distribution.

Based on this brute-force approach, the first practical external memory suffix tree construction algorithm was developed in [18]. Hunt et al.'s incremental construction trades an ideal $O(N)$ performance for locality of access to the tree during its construction. The output tree is in fact represented as a forest of several suffix trees. The suffixes in each such tree share a common prefix. Each tree is built independently and requires scanning of the entire input string for each such prefix. The idea is that the suffixes that have prefix, say, aa fall into a different subtree than those starting with ab, ac and ad. Hence, once the tree

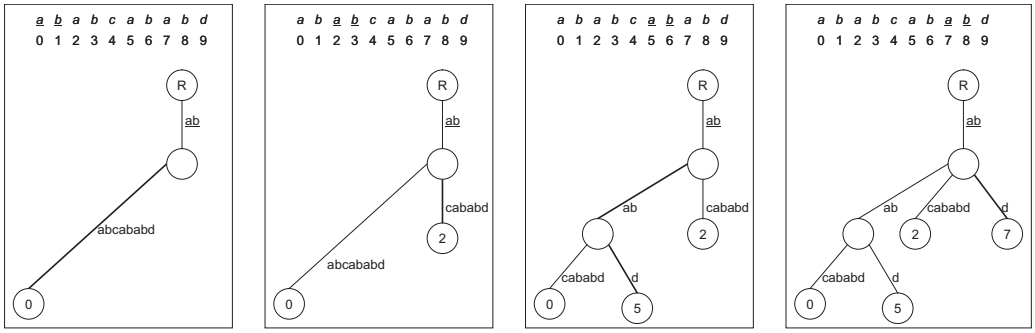

Figure 10. The steps of building the sub-tree for prefix ab and input string $X = ababcababd$ with the algorithm by Hunt et al.[18]

for all suffixes starting with aa is built, it is never accessed again. The tree for each prefix is constructed independently in main memory, and then is written to disk.

The number of partitions p is computed as the ratio of the space required for the tree of the entire input string, ST_{total}, to the size of the available main memory M, i.e. $p = |ST_{total}|/M$. Then, the length of the prefix for each partition can be computed as $\log_{|\Sigma|} p$, where $|\Sigma|$ is the size of the alphabet. This works well for non-skewed input data but fails if for a particular prefix there is a significantly larger amount of suffixes. This is often the case in DNA sequences with a large amount of repetitive substrings. In order to fit a tree for each possible prefix into main memory, we can increase the length of the prefix. This, in turn, exponentially increases the total number of partitions, and therefore, the total number of input string scans.

The construction of the sub-tree for prefix ab and input string $X = ababcababd$ is shown in Figure 10. Note that the sub-tree is significantly smaller than the suffix tree for the entire input string. The pseudocode is given in Figure 9 [Bottom].

We remark that we iterate through the input string as many times as the total number of partitions. The construction of a tree for each partition is performed in main memory. At the end, the suffix tree for each partition is written to disk. Note also that in order to perform the brute-force insertion of each suffix into the tree we need to randomly access the input string X, which therefore has to reside in memory. Since the input string is at least an order of magnitude smaller than the tree, this method efficiently addresses the problem of random accesses to the tree in secondary storage, but cannot be extended to inputs which are larger than the main-memory instantiation for holding X.

The algorithm performs much faster than the *TOP-Q* algorithm, despite the fact that its internal time is quadratic in the length of the input string. This is because for p partitions all p passes over the input string are performed in main memory, and the tree is traversed in main memory as well. Thus, the algorithm performs only $O(p)$ random accesses: namely, when writing the tree for each partition. For the Human DNA of size up to 247 MB (Human chromosome I) input, the suffix tree with Hunt et al.'s algorithm can be constructed in 97 minutes [36] compared to *TOP-Q* with 96 hours for the same input on the same machine.

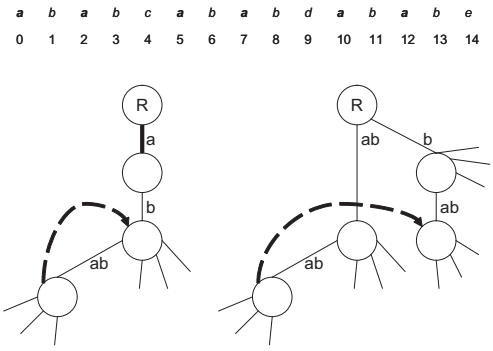

Figure 11. Difference between the sparse suffix links [**Left**] and the traditional suffix links [**Right**].

The performance of Hunt et al.'s algorithm degrades drastically if the input string does not fit the main memory and should be kept on disk. In this case we have $O(pN)$ random accesses, this time to the input string.

2.3. Distributed and paged suffix trees

A similar idea of processing suffixes of X separately for each prefix was developed in [7, 8]. The distributed and paged suffix tree (*DPST*) by Clifford and Sergot [7], which was proposed first in context of distributed computation, has all the properties to be efficiently implemented to run using external memory. As before, the suffixes of X are grouped by their common prefix whose length depends on the size N of X and the amount of the available main memory. The number of suffixes in each subtree is small enough for the tree to be entirely built in main memory. Therefore, random disk access to the sub-tree during its construction is avoided. The main difference from Hunt et al.'s algorithm of the previous section is that the sub-tree for each particular prefix is built in an asymptotic time *linear* in N and not quadratic. In order to do so, the *DPST* algorithm uses the idea developed in [1] to build the suffix tree on words. The main ideas in [1] are similar to the Ukkonen algorithm [37] described in Section 2.1. However, the Ukkonen algorithm relies heavily on the fact that *all* suffixes of X are inserted, whereas the suffix tree on words is built only for some suffixes of X, namely the ones starting at positions marked by delimiters.

DPST applies this idea considering the particular prefix as the delimiter for producing the sub-tree for this prefix. It introduces the idea of *sparse suffix links* (SSL) instead of regular suffix links. A SSL in a particular subtree leads from each internal node v_i with incoming path label w to another internal node v_j in the same sub-tree whose incoming path-label corresponds to the largest possible suffix of w found in the same sub-tree (or to the root if the largest such suffix is an empty string).

We explain the difference between the sparse suffix link and the regular suffix link in the following example. Suppose we have a sub-tree for a prefix a for $X = ababcababdababe$ (see

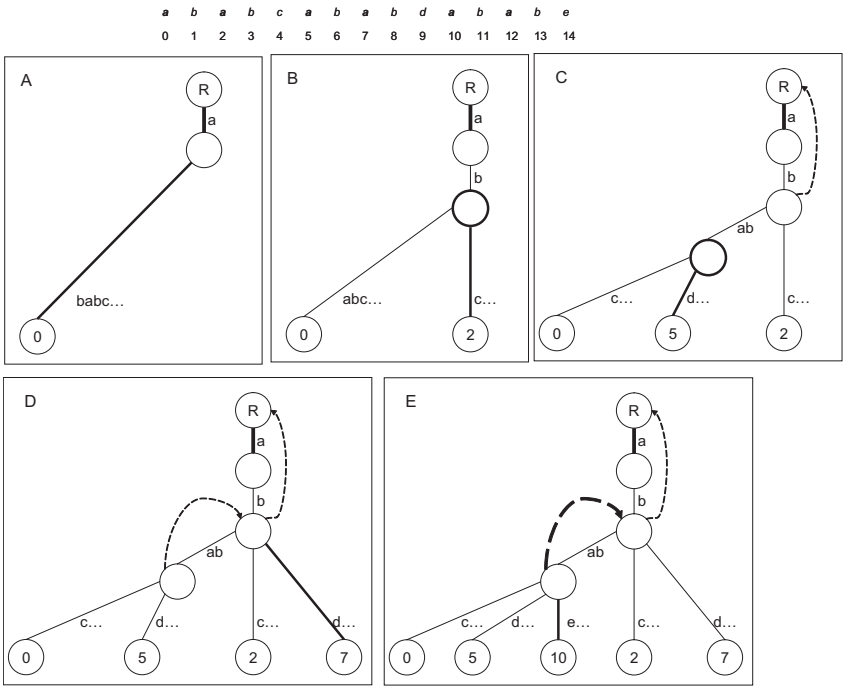

Figure 12. Steps of the construction of the sub-tree for prefix a by the distributed and paged suffix tree construction algorithm of Clifford and Sergot. Input string $X = ababcababdababe$.

Figure 11). In the regular suffix tree, the suffix link from the internal node with an incoming path label $abab$ leads to the node with the incoming path-label bab. However, in the sub-tree for prefix a, there is no suffix starting with bab. So the longest suffix of $abab$ which can be found in this sub-tree is ab, and the SSL leads to the internal node with the incoming path label ab.

Let us follow an example for the sub-tree construction for $X = ababcababdababe$ and prefix a in Figure 12. This sub-tree will contain only the suffixes of X starting at positions $0, 2, 5, 7, 10, 12$. Thus, we need to insert only these six suffixes to the tree. First, we insert suffix S_0 by creating leaf L_0. Next, we add S_2 by finding that $X[1] = X[3]$ and $X[2] \neq X[4]$. We split an edge and add leaf L_2. Now it is the turn for suffix S_5. Since the first four characters of S_5 correspond to some path in the tree, but $X[9] = d$ does not. Therefore, we add leaf L_5 and create an internal node with incoming path label $abab$. We see that the longest suffix of $abab$ in this sub-tree is ab. We create a sparse suffix link from internal node for $abab$ (marked by $\star\star$ in Figure 13) to the one for ab (marked by $\star$). When we create a new leaf out of the $\star\star$-node for suffix S_{10}, we follow the SSL and create the same e-child from the $\star$-node (Figure 13).

The use of these sparse suffix links for adding new leaves to the sub-tree allows to perform the construction of each sub-tree in time linear in N. The $DPST$ runs in time $O(NP)$ where

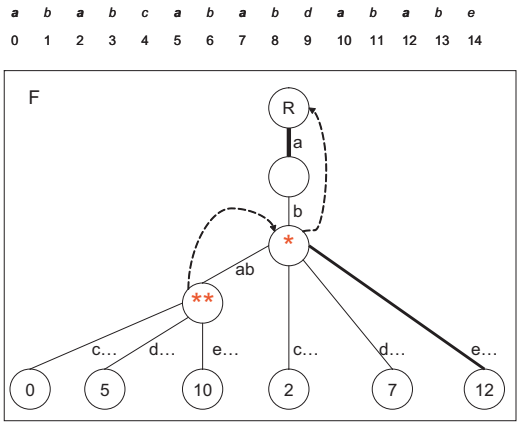

Figure 13. Sample output of the *DPST* algorithm by Clifford and Sergot.

P is the total number of different prefixes. Despite the superior asymptotic internal running time w.r.t. the previous algorithm, the practical performance and the scalability of the *DPST* as implemented in [7] were inferior to the program by Hunt et al. [18] for the real DNA data used in the experiments.

2.4. Top Down Disk based suffix tree construction (TDD)

Quadratic in the worst case, but a more elaborated approach of the Top Down Disk based suffix tree construction algorithm (*TDD*)[36] takes the performance of the on-disk suffix tree construction to the next level. The base of the method is the combination of the *wotdeager* algorithm of Giegerich et al. [16] and Hunt et al.'s prefix partitioning described above. Being still an $O(N^2)$ brute-force approach, *TDD* manages more efficiently the memory buffers and is a cache-conscious method which performs very well for many practical inputs.

The first step of *TDD* is the partitioning of the input string in a way similar to that of the algorithm by Hunt et al.. Now, the tree for each partition is built as follows. The suffixes of each partition are first collected into an array where they are represented by their start positions. Next, the suffixes are grouped by their first character into *character groups*. The number of different character groups gives the number of children for the current tree node. If for some character there is a group consisting only of one suffix, then this is a leaf node and is immediately written to the tree. If there is more than one suffix in the group, the LCP of all the suffixes is computed by sequential scans of X from different random positions, and an internal node at the corresponding depth is written to the tree. After advancing the position of each suffix by $|LCP|$, the same procedure as before is repeated recursively. The pseudocode of the *TDD* algorithm is given in Figure 14.

To illustrate the algorithm, let us observe several steps of the *TDD* suffix tree construction which are depicted in Figure 15. Suppose that we have partitioned all the suffixes of X by

***TDD* algorithm**
for each *prefix PR* of length *prefix_len*
collect suffixes starting with *PR* into array
sort suffixes by the first character
output groups with 1 suffix as leaf nodes of the tree
push groups with more than 1 suffix **into the stack**
while stack is not empty
pop suffixes of the same group **from the stack**
find *LCP* of all suffixes in the group by sequential character comparisons
output internal node at the depth *LCP*
advance position of each suffix by *LCP*
sort suffixes by the first character
output groups with 1 suffix as leaf nodes of the tree
push groups with more than 1 suffix into the stack

Figure 14. Pseudocode of the *TDD* algorithm [36].

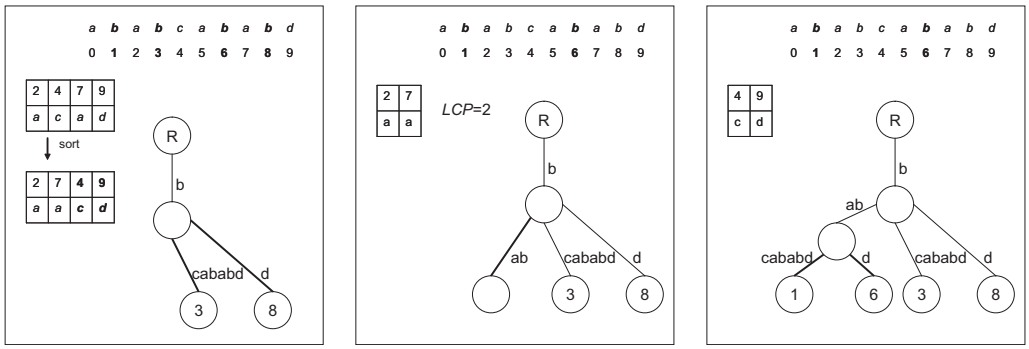

Figure 15. The steps of the TDD algorithm [36] for building the sub-tree for prefix b and input string $X = ababcababd$.

a prefix of length 1. This gives four partitions: a, b, c and d. We show how *TDD* builds the suffix tree for partition b. The start positions of suffixes starting with b are $\{1, 3, 6, 8\}$. Since the prefix length is 1, the characters at positions $\{2, 4, 7, 9\}$ are sorted lexicographically. This produces three groups of suffixes: a-group: $\{2, 7\}$, c-group: $\{4\}$ and d-group: $\{9\}$. Since the c-group and d-group contain one suffix each, the suffixes in these groups produce leaf nodes and are immediately added to the tree. The a-group contains two suffixes, and is therefore a branching node. $|LCP_{2,7}| = 2$, and therefore the length of the child starting with a equals 2. At this depth, the internal node branches at positions $\{4, 9\}$, which after sorting result into two leaf nodes: the children starting with c and d respectively.

The main distinctive feature of the *TDD* construction is the order in which the tree nodes are added to the output tree. Observe that the tree is written in a top-down fashion, and the nodes which were expanded in the current iteration are not accessed anymore. This reduces

the number of random accesses to the partially built tree and the new nodes can be written directly to the disk. The number of random disk accesses is $O(P)$ as in Hunt et al.'s algorithm. However, the size of each partition may be much bigger than before since now the main memory buffer for the suffix tree data structure does not have to hold an entire sub-tree.

This pattern of accessing the tree was shown to be very efficient for cached architectures of the modern computer. It was even shown that the *TDD* algorithm outperforms the linear-time algorithm by Ukkonen for some inputs in case when all the data structures fit the main memory. For the same input of 247 millions of symbols (Human chromosome I) [44], which took about 97 minutes with the suffix-by-suffix insertion of Hunt, *TDD* builds the tree in 18 minutes [36].

As before, the algorithm performs massive random accesses to the input string when it does the character-by-character comparisons starting at different random positions. The input string for the *TDD* algorithm cannot be larger than the main memory.

Another problem of *TDD* is the suffix tree on-disk layout. The trees for different partitions are of different sizes, and some of them can be significantly bigger than the main memory. This poses some problems when loading the subtree into main memory for querying. If the entire subtree cannot be loaded into and traversed in the main memory, the depth first traversal of such a tree requires multiple random accesses to different levels of on-disk nodes.

2.5. The partition-and-merge strategy of Trellis

The oversized subtrees caused by data skew can be eliminated by using set of different-length prefixes, as shown in [31]. In practice, the initial prefix size is chosen so that the total number of prefixes P will allow to process each of the P sub-trees in main memory. For example, we can hold in our main memory in total T_{max} suffix tree nodes. The counts in each group of suffixes sharing the same prefix are computed by a sequential scan of input string X. If a count exceeds T_{max}, then we re-scan the input string from the beginning collecting counters for an increased prefix length. Based on the final counts, none of which exceeds T_{max}, the suffixes are combined into approximately even-sized groups. As an example consider the case when suffixes starting with prefix *ab* occur twice more often than the suffixes starting with *ba* and *bb*. We can combine suffixes in partitions *ba* and *bb* into a single partition *b* with approximately the same number of suffixes as contained in partition *ab*. The maximum number of suffixes in each prefix partition is chosen to ensure that the size of the tree for suffixes which share the same prefix will never exceed the main memory. This is in order to ensure that each such subtree can be built and queried in main memory.

Based on this new partitioning scheme, Phoophakdee and Zaki [31] proposed another method for creating suffix trees on disk – the *Trellis*[‖] algorithm. The main innovative idea of this method is the combination of the prefix partitioning and the horizontal partitioning of the input into consecutive substrings, or *chunks*. In theory, the substring partitioning does not work for any input, since the suffixes in each substring partition do not run till the end of

[‖] *Trellis* stands for **E**xternal **S**uffix **TR**ee with Suffix **L**inks for **I**ndexing Genome-Sca**L**e Sequences.

```
TRELLIS algorithm
partition X into k substrings
for each substring X_i
        build suffix tree ST_X_i
        for each prefix PR in collection of variable-length prefixes
                find sub-tree starting with PR
                write this sub-tree into a separate file on-disk

for each prefix PR in collection of variable-length prefixes
        load from disk all sub-trees starting with PR
        merge sub-trees into 1 sub-tree for prefix PR
        write this sub-tree back to disk
```

Figure 16. Pseudocode of the *Trellis* algorithm.

the entire input string. However, this horizontal partitioning works for most practical inputs. Consider for example the Human genome sequence of about 3 GB in length. In fact, there is not a single string representing Human genome, but rather 23 sequences of DNA in 23 different Human chromosomes, with the largest sequence being only about 247 MB in size. Those chromosome sequences represent natural partitions of the entire genome.

If the size of each natural chunk of the input does not allow us to build the suffix tree for it entirely in main-memory, the chunk can be split into several slightly overlapping substrings. We append to the end of each such substring except the last one, a small "tail", 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 tree of the partition. In practice, for real-life DNA sequences, the length of such a tail is negligeably small compared to the size of the partition itself.

After partitioning the input into chunks of appropriate size, *Trellis* builds an independent suffix tree for each chunk. It does not output the entire suffix tree to disk, but rather writes to disk the different sub-trees of the in-memory tree. These sub-trees correspond to the different variable-length prefixes. Once trees for each chunk are built and written to disk, *Trellis* loads into memory the subtrees for all the chunks which share the same prefix. Then it merges these subtrees into the shared-prefix-based subtree for an entire input string. The pseudocode of the *Trellis* algorithm is shown in Figure 16.

As an example, let us apply the *Trellis* method to our input string $X = ababcababd$. Let the collection of prefixes for a prefix-based partitioning be $\{ab, ba, c, d\}$. Next, we partition X into two substrings $X_1 = abab$ with "tail" c, and $X_2 = cababd$. Note the overlapping symbol c which is used as a sentinel for suffixes of X_1. We build in memory the suffix tree for X_1, which is shown in Figure 17 [A], and we output it to disk in the form of two different subtrees: one for prefix ab and the second for prefix ba. The same procedure is performed for X_2 (Figure 17 [B]). Then, we load into main memory the subtrees for, say, prefix ab and we merge those sub-trees into the common ab-subtree for the entire X.

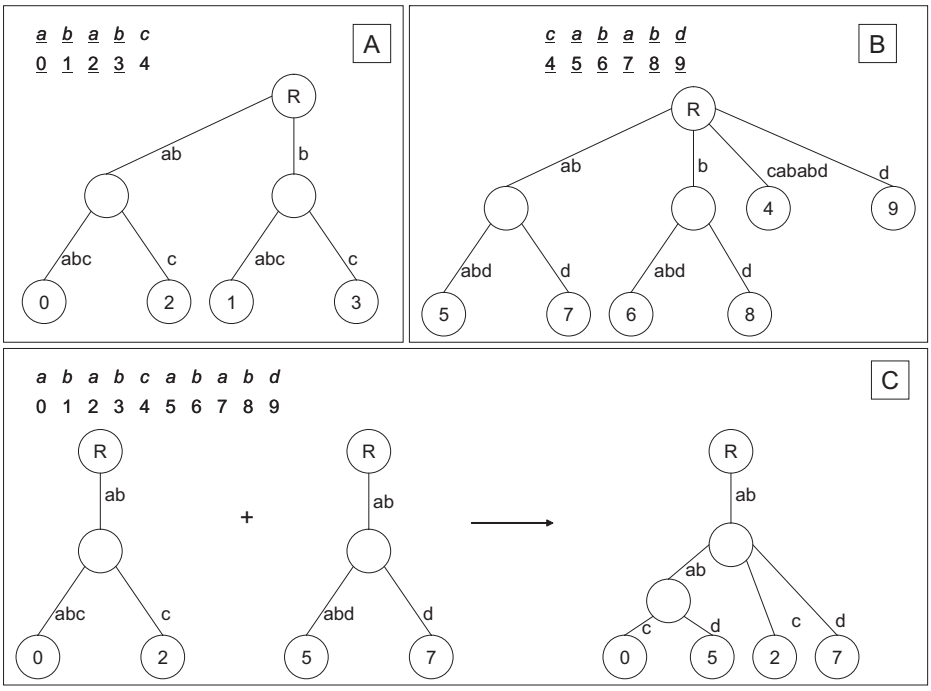

Figure 17. The steps of the *Trellis* algorithm applied to input string $X = ababcababd$. **A**. Building the suffix tree for substring $X_1 = abab(c)$. **B**. Building the suffix tree for substring $X_2 = cababd$. **C**. Merging the sub-trees for prefix ab. The total size of the tree structures at each step allows to perform each step in main memory.

The merge of subtrees for different chunks is performed by a straightforward character-by-character comparison, which leads to the same $O(N^2)$ worst-case internal time as the brute force algorithms described above.

Trellis was shown to perform at speed comparable to *TDD*. Further, *Trellis* does not fail due to insufficient main memory (for holding the trees for each chunk or the subtrees with a common prefix).

If we have K chunks and P prefixes in the variable-length prefixes collection, the number of random disk accesses is $O(KP)$. Since both K and P depend on the length of the input string N, the execution time of *Trellis* grows quadratically with the increase of N, and is therefore not scalable for larger inputs.

During the character-by-character comparison in the merge step, the input string is randomly accessed at different positions all over the input string. Therefore, the scalability of *Trellis* does not go beyond the size of the main memory designated for the input.

2.6. DiGeST and an external memory multi-way merge sort

A simple recent approach to construct suffix trees is based on an external memory multi-way merge sort [14]. The *DiGeST*** algorithm proposed in [3] performs at a speed comparable with *TDD* and *Trellis*, and scales for larger inputs since it does not use prefix-based partitioning, but rather outputs a collection of small suffix trees for the different sorted lexicographic intervals.

As in *Trellis*, *DiGeST* first partitions the input string into k chunks. The suffixes in each chunk are sorted using any in-memory suffix sorting algorithm (for example [25]). The suffix array for each chunk is written to disk. To each position in this suffix array a short prefix of the suffix is attached. These prefixes significantly improve the performance of the merging phase.

After sorting the suffixes in each chunk, consecutive pieces of each of the k suffix arrays are read from the disk into input buffers. As in the regular multi-way merge sort, 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, it is emptied to disk. In order to determine the order of suffixes from different input chunks, we first compare the prefixes attached to each suffix start position. Only if these prefixes are equal, we compare the rest of the suffixes character-by-character. This comparison requires that the input string be kept in main memory.

Due to the character-by-character comparison of the suffixes, *DiGeST* runs in $O(N^2)$ internal time. Recall that on average the performance is $O(N \log N)$. The same comparison is performed in order to calculate the longest common prefix of the current suffix with the last suffix previously inserted into the tree. The calculated $|LCP|$ determines the place where the internal node is created, and a new leaf for each suffix is added as a child of this internal node. In this way we build the suffix tree in the output buffer.

Before writing the output buffer to disk, the lexicographically largest suffix in this tree is added to a collection of "dividers" which serve locating multiple trees on disk. Since the output buffer is of a pre-calculated size, all trees are of equal size, and thus, the problem of data skew is completely avoided. Further, each tree is small enough to be quickly loaded into the main memory to perform a search or comparative analysis. The pseudocode of *DiGeST* is given in Figure 18.

While *DiGeST* still requires the input string to be in main memory, from an external memory point of view, it is very efficient: the algorithm performs only two scans over the disk data and furthermore accesses the disk mainly sequentially. From an internal running time point of view, this algorithm still belongs to the group of brute-force algorithms with a quadratic running time.

** *DiGeST* stands for **Di**sk-based **Ge**neralized **S**uffix **T**ree.

```
DiGeST algorithm
        partition X into k substrings
        for each substring X_i
                build suffix array SA_X_i
                write SA_X_i to disk
        Merge( )

Merge( )
        allocate k input buffers and 1 output buffer
        for each input buffer
                load part of SA_X_i into input buffer
        create heap of size k
        read first element of each input buffer into a heap

        while heap is not empty
                transfer the smallest suffix of substring j
                        from the top of heap into output buffer
                find LCP of this suffix with the last inserted suffix
                create a new leaf in the output suffix tree
                if output buffer is full
                        write it to disk

                if input buffer j is empty
                        if not end of SA_X_i
                                fill input buffer j with the next suffixes
                if input buffer j is not empty
                        insert next suffix from input buffer j into heap
```

Figure 18. Pseudocode of the *DiGeST* algorithm.

2.7. Suffix tree on disk layouts

We remark that most of the algorithms described above [7, 18, 36, 31, 3] do not deliver a single suffix tree on disk, but rather a forest of suffix trees. This is useful from both the construction and the query points of view. Regarding the query efficiency, if a single suffix tree is of a size much larger than the available main memory, then searching for a pattern of length q may incur q random I/Os plus one random I/O to collect each occurrence by reaching the corresponding leaves. The need to partition the tree into meaningful partitions is even more prominent for algorithms which require a depth-first traversal (DFS) of the entire tree, such as finding a longest common substring, or finding the total number of all different substrings. In these cases, the number of random I/Os will be $O(N)$, and the performance of DFS-based algorithms will severely degrade.

Thus, important practical requirements for the output trees are that each tree can be sequentially loaded and traversed entirely in main memory and that each tree has some unique identifier to be located quickly.

The most accepted scheme for tree partitioning is partitioning by prefix. For each prefix there is a separate tree which contains all the suffixes sharing this prefix. The collection of all possible prefixes of length p is of size $P = O(|\Sigma|^p)$. Note that, in order to search for a pattern, we need to find the corresponding prefix, load the corresponding sub-tree by one sequential read and then find all the occurrences of this pattern in this sub-tree. For the DFS, we read

and analyze in main memory entire sub-trees, so the maximum number of random disk I/Os equals the total number P of such sub-trees.

Constant-size prefixes are used in the algorithm by Hunt et al. [18] and in the TDD algorithm [36]. This partitioning scheme works well except when the real life input data is so skewed that for some prefixes the trees are very small, whereas for others so large that they can not be entirely held in the available main memory.

Partitioning by the variable-length prefixes is used in $Trellis$ [31] to solve the data skew problem and is described in detail in Section 2.5. The search follows the same pattern as before.

$DiGeST$ partitions the trees by lexicographic intervals of the suffixes. The 32-bit prefix of the smallest and of the largest suffix in each tree is stored in the collection of dividers, and the search starts by locating the proper interval and then loading into memory the entire tree corresponding to this interval. All the trees are of equal size. For exact and approximate pattern search the partitioning by intervals works quite well.

2.8. Summary

The main features of the practical EM algorithms for the suffix tree construction and some performance and scalability benchmarks are summarized in the table of Figure 19. The fastest and the most scalable algoritms [3, 31, 36] are able to build the suffix tree for up to 7GB of genomic data in a matter of several hours on a single machine.

3. The remaining challenges

Despite these impressive results, practicality of the suffix tree construction algorithms for massive string data sets is not yet achieved.

The largest challenge is to build suffix trees for strings larger than the main memory in a reasonable time. The performance of all algorithms described above degrades drastically once the input string outgrows the main memory and has to be accessed on disk. Note that building suffix tree indexes for strings that do not fit the main memory is of big importance, because, after all, if the input string is entirely in the main memory, then it might be possible to design on-line searching algorithms which are faster than algorithms using disk-based indexes.

One of the first attempts to address this input-string memory bottleneck was undertaken by the authors of *Trellis* with the improved method *Trellis* with string buffer (*Trellis+SB*)

¶These sample performances give an order of magnitude but are not directly comparable since they were obtained using different machines (2GB of main memory each.)

‖The time for TDD was obtained by using a DNA encoding with 4 bits per character, since 3GB of DNA do not fit the main memory of 2GB [36]. *Trellis* and *DiGeST* use a 2-bits-per-DNA-character encoding and do not include the non-DNA characters such as "N" which stands for the undefined DNA symbol. DiGeST maps the new positions after "slicing out" the unknown characters to the original positions in the raw sequence. Note that if we index all the characters which appear in genomic sequences, we will face the problem of invalid common substrings of significant length which consist of long stretches of "N"s and do not represent actual common substrings.

Method name	Underlying main memory algorithm	Asymptotic internal running time	Max. number of random disk I/Os	Sample performance[¶] for ≈ 50MB of Human chromosome I	Scalability: (the largest input handled with 2GB of RAM) [‖]
TOP-Q [4]	Suffix tree construction by Ukkonen [37]	N	N	7 h [31]	≈ 50MB: 4 hours [4]
DPST [7]	Suffix tree on words [1] for each partition	$N \times P$	P	Not reported	≈ 187MB: time not reported
Hunt et al. [18]	Brute force for each partition	$N^2 \times P$	P	12 min [36]	≈ 256MB: 13 h [18]
TDD [36]	*Wotdeager* [16] for each partition	$N^2 \times P$	P	2 min [36]	≈ 3GB: 30 h [36]
Trellis [31]	Ukkonen [37] for each partition, brute force for merge	$N + (\frac{N}{P})^2$	$K + KP$	2 min [31]	≈ 3GB: 4 h [31]
DiGeST [3]	Suffix array [25] for each partition, multi-way merge sort for merge [14]	$N \log N + + \frac{N^2}{K}$	$K + K^2$	2 min [3]	≈ 7GB: 6 h [3]

Figure 19. The key features of the practical algorithms for the external memory suffix tree construction. Here, N denotes the total input size, P the number of partitions by common prefixes, K the number of substring partitions.

[32]. The authors designed a string buffer which keeps the characters that are more likely to be accessed in main memory, and loads the rest from disk when needed. Since suffix-tree edges contain positions of the corresponding substring inside the input string, *Trellis+SB* replaces these positions wherever possible by positions in one representative partition. This small representative part of the input is kept in memory during the 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 allows in practice to reduce the number of accesses to the on-disk input string by 95% (for DNA sequence of the Human genome). The authors report that they were able to build the suffix tree for the Human genome using 512MB of memory in 11 hours on a modern machine. The performance for larger inputs than 3GB was not reported, maybe because the remaining 5% of an input of, say, 10GB translate into

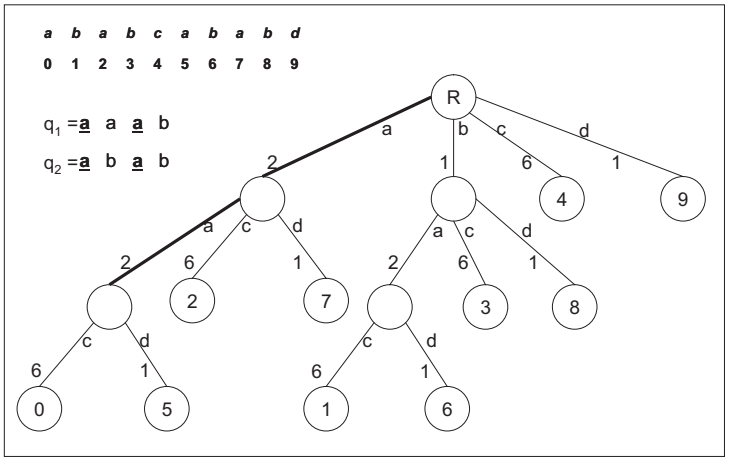

Figure 20. Disk-friendly pattern search in the suffix tree for string $X = ababcababd$. The first characters of each edge are implied by the position of a child in the array of children. The length of each edge is shown, which is deduced from the start and end positions of edge-label substrings. For query $q_1 = aaab$ we match $q_1[0]$ and $q_1[3]$, and then we retrieve the leaf L_0 as well as the substring $X[0, 3]$. Verification fails, since $aaab \neq abab$. Pattern $q_1 = aaab$ is not a substring of X. For query $q_2 = abab$ we follow the same path. This time the verification is successful, since $q_2 = X[0, 3]$. We report all the occurrences of q_2 in X by collecting leaves 0 and 5. In each case, only one random access to the input string was performed.

500 million random disk I/Os. This method works only if the representative partition can be found for the entire input which is not always the case. There is currently no general practical solution for building suffix trees for strings larger in size than the main memory.

With a look at the potential applications of the on-disk suffix trees, we notice another important problem. A 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. Let us assume that we have constructed a suffix tree for an input string significantly larger than the main memory. In this case the input string is entirely on disk. Note that, to search for a query string q in this tree by a traditional suffix tree traversal [17] we (naïvely) compare the characters of q to the characters of X 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 X is on disk. This takes in the worst case as many random accesses to the input string as the length of the query.

However, massive random access to X during a search can be avoided if we follow the PATRICIA search algorithm originally described in [28]. The outgoing edges from an internal node are indexed according to the character specified by their start positions. The search consists of two phases. In the first phase, we trace a downward path from the root of the tree to locate a leaf L_i, which does not necessarily match all the characters of query string q. We start out from the root and only compare some of the characters of q with the branching

characters found in the arcs traversed until we either reach leaf L_i, or no further branching is possible. In the latter case, we choose L_i to be any descending leaf from the last node traversed, say node v. In the second phase, we read substring $X[i, i+|q|]$ from the input string X which is on disk, by that performing only one random access to the input, instead of $|q|$ as in the usual suffix tree search. We compare $X[i, i+|q|]$ to q; if both are identical, we report an occurrence of q in X and collect all the remaining occurences from the leaf nodes below v (if v is not a leaf). Consider, for example, the suffix tree for $X = ababcababd$ and the two queries $q_1 = aaab$ and $q_2 = abab$ shown in Figure 20. By matching only the first and the third characters of q_1 or q_2, and after that verifying the queries against suffix S_0, we perform only one random access to the input string per query. This example shows a great potential for suffix trees on disk. Such an efficient (from the external memory point of view) search does not yet exist for alternative indexing structures, such as suffix arrays.

However, this type of search cannot be performed with such an efficiency if we want to find an approximate occurrence of the query string in X. The proposed algorithm for approximate matching using suffix trees [29] requires the comparison of actual characters of the string in order to find an edit distance between the different substrings of q and the substrings of edge-labels. Therefore, the approximate pattern matching algorithm proposed in [29] will not work efficiently in external memory settings in case that the input string cannot be held in main memory. This requires the development of new approaches for approximate pattern matching. The adaptability and the efficiency of other algorithms using on-disk suffix tree layouts is yet to be investigated.

The important drawback for algorithms with a satisfactory practical performance (cf. [36, 31, 3]) is the internal asymptotically quadratic time of these algorithms. Though $O(N \log N)$ on average, this time increases dramatically if we work on the suffix tree construction for similar DNA sequences. For example, in order to compare DNA sequences of different genomes of the same species, when building the generalized suffix tree for these sequences, all the above algorithms fail to perform in a satisfactory time.

The last note about differences between on-disk and in-memory suffix trees is the usefulness of suffix links. Recall that suffix links connect each internal node representing some substring αy (where α is one character long) of X with some other internal node where substring y ends. The suffix links, a by-product of the linear time construction algorithms, can be useful by themselves. An example is finding the occurrences of all substrings of the query q in the input string X. In this case, after locating some prefix $q[0, i]$ of the query string, we follow the suffix link from the lowest internal node of the path found in the tree, and check the next substring starting from the node at the other end of the suffix link. By this we save as many character comparisons as the depth of this internal node from the root. Though not used during most of the suffix tree construction algorithms presented here, suffix links can be recovered in a post-processing step [31]. We believe that these recovered suffix links in the external memory settings are of a limited use, since the link leads to the different subtree layered in distant disk locations. This means that an assumed "constant-time" jump following a suffix link causes in fact an entire random disk access. Note that finding all substrings of the query will require the same number of disk accesses using suffix links as checking all different suffixes of q using the PATRICIA search in the corresponding subtrees. As for other algorithms which make use

of suffix links (see for example [22, 23]), it seems that they might require new non-trivial adaptations when moved from the in-memory to the on-disk settings.

4. Theoretical basis for future research

In parallel to the development of practical software for the suffix tree construction on disk, promising theoretical results for this problem were obtained.

As a matter of fact, an algorithm which runs in $O(SORT(N))$ time in the theoretical DAM computational model was developed in [12]. $SORT(N)$ means that we can build the suffix tree for the input string of any size with a time complexity equal to the constant number of external memory sorts, applied to integers. This algorithm builds separate suffix trees for even and odd suffixes and then merges them into the suffix tree for X. The merge phase is the real bottleneck of this algorithm, and it is not clear if it can possibly be implemented in practice, as was also noted in [35]. The details of this external memory algorithm are extremely complex preventing, so far, an implementation of this algorithm.

More promising results were obtained for building a suffix array of X. As was shown in [12], not only can we convert each suffix tree into a suffix array in linear time by a depth-first traversal of the suffix tree, but we can also convert the suffix array into a suffix tree in linear time, assuming that the suffix array is augmented with the longest common prefix information between consecutive suffixes in the suffix array. This is performed by simulating an Euler tour of the tree under construction using the LCP information.

Following this direction, in order to develop a practical algorithm to construct suffix trees for input strings of any size, we need three essential steps to be efficient from an external memory point of view: building the suffix array, generating an array of LCPs and converting this enhanced suffix array into the suffix tree.

As for the suffix array construction, the optimal results obtained for the DAM computational model look very promising. The *SKEW* algorithm, proposed in [19] and generalized into the *DC* (Difference Cover) algorithm in [20], is a simple and elegant algorithm which builds suffix arrays on disk in $O(SORT(N))$ time. This algorithm was first implemented in the *DC-3* program [10] and has demonstrated a promising practical performance for large inputs. This algorithm can be used as a first step to overcome the input string size bottleneck on the way to fully-scalable suffix trees.

The conversion of a suffix array into a suffix tree turned out to be disk-friendly, since reads of the suffix array and writes of the suffix tree can be performed sequentially.

However, the suffix array needs to be be augmented with the LCP information in order to be converted into a suffix tree. There exist linear-time, space-efficient and easy-to-implement LCP computation algorithms (see for example [21, 26]) which, however, perform random access to at least one intermediate array of size N. These algorithms would severely degrade in performance once N is larger than the main memory, therefore they need to be modified for such a case. Theoretical results for computing the LCP in external memory settings in time $O(SORT(N))$ were presented in [6]. These results are based on range minima queries which are performed using special tree-like data structures and an external memory sort of queries to minimize random disk I/Os. These results were used by the authors of the *DC* algorithm

[20] to show how to compute the LCP enhancement for their suffix array. It is currently not clear how efficient the presented algorithm for the LCP computation would be in a practical implementation.

It may be only one step which divides us from a scalable solution for constructing suffix trees on disk for inputs of any type and size. Once this is done, a whole world of new possibilities will be opened, especially in the field of biological sequence analysis.

REFERENCES

1. A. Andersson, N.J. Larsson, and K. Swanson Suffix trees on words. *Proceedings of the CPM-1996*, LNCS, 1075: 102–115, 1996.
2. A. Apostolico, W. Szpankowski Self-Alignments in Words and Their Applications. *J. Algorithms*, 13(3): 446–467, 1992.
3. M. Barsky, U. Stege, A. Thomo, C. Upton A new method for indexing genomes using on-disk suffix trees. *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM*: 649–658, 2008.
4. S.J. Bedathur and J.R. Haritsa Engineering a fast online persistent suffix tree construction. *Proceedings of the 20th International Conference on Data Engineering*: 720, 2004.
5. D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and D. Wheeler GenBank. *Nucleic Acids Research*, 34: 2006.
6. Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, and J. Vitter External-Memory Graph Algorithms. *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*: 1995.
7. R. Clifford, M. J. Sergot Distributed and Paged Suffix Trees for Large Genetic Databases. *Proceedings of the CPM-2003*: 70–82, 2003.
8. R. Clifford Distributed suffix trees. *J. Discrete Algorithms*, 3(2–4): 176–197, 2005.
9. A. Crauser, P. Ferragina A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory. *Algorithmica*, 32(1): 1–35, 2002.
10. R. Dementiev, J. Kärkkäinen, J. Mehnert, P. Sanders Better external memory suffix array construction. *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*: 2005.
11. M. Farach and S. Muthukrishnan Optimal Logarithmic Time Randomized Suffix Tree Construction. *Proceedings of the 23rd international Colloquium on Automata, Languages and Programming, LNCS*, 1099: 550-561, 1996.
12. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan On the sorting-complexity of suffix tree construction. *J. of the ACM*, 47(6): 987-1011, 2000.
13. J. Fischer, V. Mäkinen, and G. Navarro An(other) Entropy-Bounded Compressed Suffix Tree. *Proceedings of the CPM-08*, LNCS 5029: 152–165, 2008.
14. H. Garcia-Molina, J. D. Ullman, J. D. Widom Database System Implementation. Prentice-Hall, Inc, 1999.
15. R. Giegerich and S. Kurtz From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19 (3): 331–353.
16. R. Giegerich, S. Kurtz, and J. Stoye Efficient Implementation of Lazy Suffix Trees. *Software—Practice and Experience*, 33(11): 1035–1049, 2003.
17. D. Gusfield Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, 1997.
18. E. Hunt, M.P. Atkinson, R.W. Irving A database index to large biological sequences. *The VLDB Jornal*, 7(3): 139–148, 2001.
19. J. Kärkkäinen and P. Sanders Simple Linear Work Suffix Array Construction *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*: 2003
20. J. Kärkkäinen, P. Sanders, S. Burkhardt Linear work suffix array construction. *J. of the ACM*, 53 (6): 918–936, 2006.
21. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. *Proceedings of the CPM-2001*, LNCS, 2089: 181–192, 2001.

22. S. Kurtz, and C. Schleiermacher REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15: 426–427, 1999.
23. S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, S. Salzberg Versatile and open software for comparing large genomes. *Genome Biol*, 5(R12): 2004.
24. D. Knuth The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998: Section 5.4: External Sorting, pp.248-379.
25. 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.
26. G. Manzini Two Space Saving Tricks for Linear Time LCP Array Computation. *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, LNCS, 3111: 372–383, 2004.
27. E. M. McCreight A Space-economical Suffix Tree Construction Algorithm. *J. of the ACM*, 23(2): 262–272, 1976.
28. D. Morrison PATRICIA – practical algorithm to retrieve information coded in alphanumeric. *J. of the ACM*, 15(4): 514–534, 1968.
29. G. Navarro and R. Baeza-Yates A Hybrid Indexing Method for Approximate String Matching. *J. of Discrete Algorithms*, 1(1): 205–209, 2000.
30. M. Nelson Fast String Searching With Suffix Trees. http://marknelson.us/1996/08/01/suffix-trees/
31. B. Phoophakdee and M. J. Zaki Genome-scale Disk-based Suffix Tree Indexing. *ACM SIGMOD International Conference on Management of Data*, 2007.
32. B. Phoophakdee and M. J. Zaki Trellis+: An Effective Approach for Indexing Massive Sequence. *Pacific Symposium on Biocomputing*, 2008.
33. L. Russo, G. Navarro, and A. Oliveira Dynamic Fully-Compressed Suffix Trees. *Proc. CPM'08*, LNCS 5029: 191–203, 2008.
34. K. Sadakane Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.*, 41(4): 589–607, 2007.
35. W. Smyth Computing Patterns in Strings. Addison-Wesley, 2003
36. Y. Tian, S. Tata, R. Hankins, J. Patel Practical methods for constructing suffix trees. *The VLDB Journal*, 14(3) : 281–299, 2005.
37. E. Ukkonen On-line construction of suffix trees. *Algoritmica*, 14(3): 1995.
38. N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen Engineering a Compressed Suffix Tree Implementation. *Proceedings of the 6th Workshop on Experimental Algorithms*, LNCS, 4525: 217–228, 2007.
39. N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen Compressed Suffix Tree – A Basis for Genome-scale Sequence Analysis. *Bioinformatics*, 23: 629–630, 2007.
40. J. Vitter and M. Shriver Algorithms for parallel memory: Two-level memories. *Algorithmica*, 12: 110–147, 1994.
41. J. Vitter External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2): 209–271, 2001.
42. P. Weiner Linear pattern matching algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*: 1–11, 1973.
43. NCBI Genebank overview: http://www.ncbi.nlm.nih.gov/Genbank/
44. Homo sapiens Chromosome 1 (36.3). http://www.ncbi.nlm.nih.gov/mapview/maps.cgi?taxid=9606chr=1