The Virtual Suffix Tree: An Efficient Data Structure for Suffix Trees and Suffix Arrays*

Jie Lin, Yue Jiang, and Don Adjeroh

Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV 26506
jlin@mix.wvu.edu, yue@csee.wvu.edu, don@csee.wvu.edu

Abstract. We introduce the VST (virtual suffix tree), an efficient data structure for suffix trees and suffix arrays. Starting from the suffix array, we construct the suffix tree, from which we derive the virtual suffix tree. The VST provides the same functionality as the suffix tree, including suffix links, but at a much smaller space requirement. It has the same linear time construction even for large alphabets, Σ, requires $O(n)$ space to store (n is the string length), and allows searching for a pattern of length m to be performed in $O(m \log |\Sigma|)$ time, the same time needed for a suffix tree. Given the VST, we show an algorithm that computes all the suffix links in linear time, independent of Σ. The VST requires less space than other recently proposed data structures for suffix trees and suffix arrays, such as the enhanced suffix array [1], and the linearized suffix tree [16]. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form.

1 Introduction

The suffix tree is an important data structure used to represent the set of all suffixes of a string. The suffix tree is efficient in both time and space, and has been used in a variety of applications, such as pattern matching, sequence alignment, the identification of repetitions in genome-scale biological sequences, and in data compression. Various algorithms have been developed for efficient construction of suffix trees [28,22,27,8]. However, one major problem with the suffix tree is its practical space requirement. The suffix array is a related data structure, which was originally introduced in [21] as a space-efficient alternative to the suffix tree. The suffix array simply provides a listing of all the suffixes of a given string in lexicographic order. The suffix array can be used in most (though, not all) situations where a suffix tree can be used.

Although the theoretical space complexity is linear for both data structures, typically, for a given string T of length n, the suffix array requires about three to five times less space than the suffix tree. The construction time for both algorithms is also $O(n)$ on average. For suffix arrays, construction algorithms that run in $O(n \log n)$ worst case[1] are relatively easy to develop, but $O(n)$ worst case algorithms are much harder to come by. Recent suffix sorting algorithms with worst-case linear time have been reported in [13,18,17,3]. Gusfield [11] provides a comprehensive treatment of suffix trees and its applications. Puglisi et al [26] provide a recent survey on suffix arrays. Adjeroh et al (2008) provide an extensive discussion on the connection between the Burrows-Wheeler transform [6] and suffix trees and suffix arrays.

For small alphabet sizes, the suffix tree and the suffix array have about the same complexity in pattern matching. For pattern matching, the suffix array requires time

* Partly supported by a DOE CAREER award.

[1] All logarithms are to base 2, unless otherwise stated.

Jie Lin, Yue Jiang, Don Adjeroh: The Virtual Suffix Tree: An Efficient Data Structure for Suffix Trees and Suffix Arrays, pp. 68–83.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1

in $O(m \log n)$ to locate one occurrence of a pattern of length m in T. However, with additional data structures, such as the `lcp` array, this time can be reduced to $O(m + \log n)$. With the suffix tree, the same search can be performed in $O(m)$ time. The problem, however, is for sequences with large alphabets. Here, $|\Sigma|$, the alphabet size is no longer negligible. Using the array representation of nodes in the suffix tree will require $O(n|\Sigma|)$ space for the suffix tree, and $O(m)$ time for pattern matching. For linear space, the linked list or binary search tree can be used, but the search time becomes $O(m|\Sigma|)$ or $O(m \log |\Sigma|)$ respectively.

The challenge therefore is to develop space-efficient data structures that can support pattern matching using the same time complexity as suffix trees, but at a practical space requirement that approaches that of the suffix array. Such a data structure should also support the complete functionality of the suffix tree, such as support for suffix links, as may be required in certain applications. Two recent data structures that have tried to address this problem are the ESA – *enhanced suffix array* [1], and the LST – *linearized suffix tree* [15,16]. Both methods are based on the notion of `lcp`-intervals [14], constructed using the suffix array and the `lcp` array. Other related data structures that have been proposed include the suffix cactus [12], suffix vectors [23,25], compact suffix trees [20], the lazy suffix trees [9], level-compressed suffix trees [4], compressed suffix trees [24], and compressed suffix arrays [10]. See also [2].

1.1 Main results

We introduce another data structure, *the virtual suffix tree* (VST), an efficient data structure for suffix trees and suffix arrays. The VST does not use the `lcp`-intervals, but rather exploits the inherent nature of the suffix tree topology. We state our main results in the form of two theorems about the VST.

Theorem 1. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, and the virtual suffix tree for T, we can count the number of occurrences of a pattern $P = P[1..m]$ in T in $O(m \log |\Sigma|)$ time, and locate all the η_{occ} occurrences of P in T in $O(m \log |\Sigma| + \eta_{occ})$ time.*

Theorem 2. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, the virtual suffix tree, including the suffix link, can be constructed in $O(n)$ time, and $O(n)$ space, independent of Σ.*

Essentially, the VST provides the same functionality as the suffix tree, but at a much smaller space requirement. It has the same linear time construction for large $|\Sigma|$, requires $O(n)$ space to store, and allows searching for a pattern of length m to be performed in $O(m \log |\Sigma|)$ time, the same time needed for a suffix tree. To provide the complete functionality of the suffix tree, we describe a simple linear time algorithm that computes the suffix links based on the VST. Although the space needed for the VST is linear (as in suffix tree implementations using linked lists or binary trees), the practical space requirement is much smaller than that of a suffix tree. The VST requires less space than other recently proposed data structures for suffix trees and suffix arrays, such as the ESA [1], and the LST [16]. On average, the space requirement (including that for suffix arrays and suffix links) is $13.8n$ bytes for the regular VST, and $12.05n$ bytes in its compact form. This can be compared with the $20n$ bytes needed by the LST or the ESA.

1.2 Organization

The next section introduces the key notations and definitions used. In Section 3, we introduce the basic data structure and discuss the properties of the VST. Section 4 presents an improved data structure, along with algorithms for its construction. A complexity analysis on the construction and use of the VST is also presented in this section. Section 5 shows how the suffix link can be constructed on the VST. The paper is concluded in Section 6.

2 Basic notations and definitions

Let $T = T[1..n]$ be the input string of length n, over an alphabet Σ. Let $T = \alpha\beta\gamma$, for some strings α, β, and γ (α and γ could be empty). The string β is called a *substring* of T, α is called a *prefix* of T, while γ is called a *suffix* of T. The prefix α is called a proper prefix of T if $\alpha \neq T$. Similarly, the suffix γ is called a proper suffix of T if $\gamma \neq T$. We will also use $t_i = T[i]$ to denote the i-th symbol in T — both notations are used interchangeably. We use $T_i = T[i..n] = t_i t_{i+1} \cdots t_n$ to denote the i-th suffix of T. For simplicity in constructing suffix trees, we usually ensure that no suffix of the string is a proper prefix of another suffix by appending a special symbol, \$ to T, such that $\$ \notin \Sigma$, and $\$ < \sigma$, $\forall \sigma \in \Sigma$.

Given a string T, its suffix tree (ST) is a rooted tree with n leaves, where the i-th leaf node corresponds to the i-th suffix T_i of T. Except for the root node and the leaf nodes, every node must have at least two descendant child nodes. Each edge in the suffix tree represents a substring of T, and no two edges out of a node start with the same character. For a given edge, the *edge label* is simply the substring in T corresponding to the edge. We use l_i to denote the i-th leaf node. Then, l_i corresponds to T_i, the i-th suffix of T. When the edges from each node are sorted alphabetically, then l_i will correspond to $T_{SA[i]}$, the i-th suffix of T in lexicographic order.

For edge (u, v) between nodes u and v in ST, the edge label (denoted $label(u, v)$) is a non-empty substring of T. The edge length is simply the length of the edge label. The edge label is usually represented compactly using two pointers to the beginning and end of its corresponding substring in T. For a given node u in the suffix tree, its *path label*, $L(u)$ is defined as the label of the path from the root node to u. Since each edge represents a substring in T, $L(u)$ is essentially the string formed by the concatenation of the labels of the edges traversed in going from the root node to the given node, u. The *string depth* of node u, (also called its length) is simply $|L(u)|$, the number of characters in $L(u)$. The *node depth* (also called node level) of node u is the number of nodes encountered in following the path from the root to u. The root is assumed to be at node depth 0.

Certain suffix tree construction algorithms make use of *suffix links*. The notion of suffix links is based on a well-known fact about suffix trees [28,20], namely, if there is an internal node u in ST such that its path label $L(u) = a\alpha$ for some single character $a \in \Sigma$, and a (possibly empty) string $\alpha \in \Sigma^*$, then there is a node v in ST such that $L(v) = \alpha$. A pointer from node u to node v is called a *suffix link*. If α is an empty string, then the pointer goes from u to the root node. Suffix links are important in certain applications, such as in computing matching statistics needed in approximate pattern matching, regular expression matching, or in certain types of traversal of the suffix tree.

A predominant factor in the space cost for suffix trees is the number of interior nodes in the tree, which depends on the tree topology. Thus, a major consideration is how the outgoing edges from a node in the suffix tree are represented. The three major representations used for outgoing edges are arrays, linked lists, and binary search trees. While the array is simple to implement, it could require a large memory for large alphabets. However, independent of the specific method adopted, a simple implementation of the suffix tree can require as large as $33n$ bytes of storage with suffix links, or $25n$ bytes without suffix links [2].

The suffix array (SA) is another data structure, closely related to the suffix tree. The suffix array simply provides a lexicographically ordered list of all the suffixes of a string. If $SA[i] = j$, it means that the i-th smallest suffix of T is T_j, the suffix starting at position j in T. A related structure, the `lcp` array contains the length of the longest common prefixes between adjacent positions in the suffix array. Combining the suffix array with the `lcp` information provides a powerful data structure for pattern matching. With this combination, decisions on the occurrence (or otherwise) of a pattern P of length m in the string T of length n can be made in $O(m + \log n)$ time. Given the new worst-case linear-time direct SA construction algorithms, and the small memory footprint of suffix arrays, it is becoming more attractive to construct the suffix tree from the suffix array. A linear-time algorithm for constructing ST from SA is presented in [2].

3 Basic Data Structure

Starting from the suffix array, we construct an efficient data structure to simulate the suffix tree (ST). We call this structure a Virtual Suffix Tree (VST). The VST stores information about the basic topology of the suffix tree, the suffix array, and the suffix links. Thus, the VST is represented as a set of arrays that maintains information on the internal nodes of the suffix tree. The leaf nodes are not stored directly. However, whenever needed, information about any leaf node can be obtained via the suffix array. Unlike the ESA and LST, the VST neither uses the `lcp`-interval tree nor stores the `lcp` array. We call the data structure a virtual suffix tree in the sense that it provides all the functionalities of the suffix tree using the same space and time complexity as a suffix tree, but without storing the actual suffix tree. Later, we show that the VST leads to a more compact representation of suffix trees and suffix arrays. (We mention that [14] also used the term "virtual suffix tree", but for a limited form of the enhanced suffix array).

Below, we present the basic VST. This structure will require 14 bytes for each node in the VST and supports pattern matching in $O(m \log |\Sigma|)$ time, for an m-length pattern. In the next section, we present an improved data structure that reduces the space cost by eliminating the need to store edge lengths, while still maintaining $O(m \log |\Sigma|)$ time for pattern matching. We also describe a more compact structure for the VST that uses only 10 bytes for each internal node of the VST, and 5 bytes for each leaf node. Pattern matching on this compact representation will, however, be in $O(m|\Sigma|)$ time.

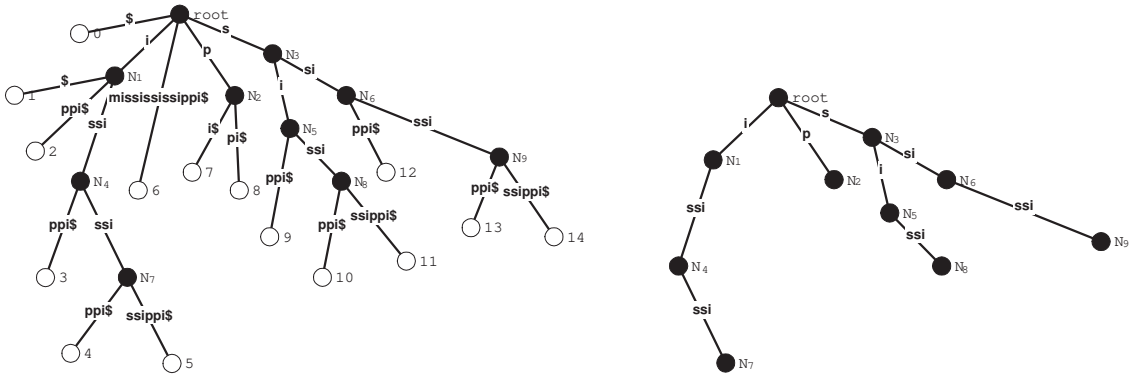
Each node in the VST corresponds to a distinct internal node in the suffix tree. In its basic form, each node in the VST is characterized by five attributes. For a given node in the VST (say node u), with a corresponding internal node in ST (say node u_{ST}), the five attributes are defined as follows.

- `sa_index`: index in the suffix array (SA index) of the leftmost leaf node under the internal node u_{ST} of the suffix tree.
- `fchild`: the node ID of the first child node of u_{ST} that is also an internal node. (Scanning is done left to right; edges at a node are also sorted left to right in ascending lexicographic order). If node u is a leaf node in the VST, the value will be negative. The absolute value will point to the first child node of the next internal node in the VST.
- `elength`: The edge length of the edge (v, u) in the VST, or equivalently (v_{ST}, u_{ST}) in the suffix tree, where v is the parent node of u and v_{ST} is the parent node of u_{ST}.
- `nfleaf`: the number of child leaf nodes *before* the first child of u_{ST} that is also an internal node.
- `nnleaf`: the number of sibling leaf nodes *after* u_{ST}, the current internal node of the suffix tree, but before the next sibling internal node.

In terms of storage, the `sa_index`, `fchild` and `elength` each requires one integer (4 bytes), while `nfleaf` and `nnleaf` each requires one byte of storage (assuming $|\Sigma| \leq 256$).

3.1 Example VST

We use an example sequence to explain the above definitions. The suffix tree and VST for the string `mississississippi$` are shown in Figure 1. Note that the string `mississississippi$` is made intentionally different from `mississippi$`, to capture some of the cases involved in a VST. Only the internal nodes (dark nodes) are explicitly stored in the VST. The leaf nodes (empty circles) are not stored. The order of storage is based on the node-depths, from top to bottom. Table 1 shows the corresponding values of the VST node attributes for each VST node in the example.



(a) (b)

Figure 1. Suffix tree and virtual suffix tree for the string $T =$ `mississississippi$`. (a) suffix tree; (b) virtual suffix tree. The number at each leaf node indicates the position in SA. The number at each internal node indicates the node ID in the VST.

3.2 Properties of the Virtual Suffix Tree

We can trace the properties of the VST based on the standard properties of a suffix tree.

node	root	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
`sa_index`	0	1	7	9	3	9	12	4	10	13
`fchild`	N_1	N_4	$-N_5$	N_5	N_7	N_8	N_9			
`elength`	0	1	1	1	3	1	2	3	3	3
`nfleaf`	1	2	2	0	1	1	1	2	2	2
`nnleaf`	0	1	0	0	0	0	0	0	0	0

Table 1. VST node attributes for the example sequence $T =$ `mississississippi$` used in Figure 1.

1. The VST only stores the internal nodes of the suffix tree. No leaf nodes in the ST are represented in the VST. Information about the leaf nodes can be obtained from the SA when needed. Then the space requirement of the VST depends on the topology of the the suffix tree, or more specifically, on the number of internal nodes.
2. The number of leaf nodes in a suffix tree is n. The number of internal nodes in the suffix tree (and hence number of nodes in the VST) is at most n.
3. The VST stores only the SA index of the leftmost leaf nodes and information about the child nodes.
4. For a given node in the VST, the number of child nodes will be no larger than $|\Sigma|$. Thus, the time needed to match a symbol is at most $O(\log |\Sigma|)$.
5. The nodes in the VST are ordered based on the internal nodes of the suffix tree using the HSAM (hierarchy sequential access method). The child nodes from any given node will be stored sequentially. The child nodes of two nearby nodes will therefore be stored in nearby locations. This is an important property for addressing problems involving locality of reference.

We introduce further definitions needed in the description below. For a given node u in the VST, we use the term *prior* node to denote the node that appears before the current node u in the HSAM ordering. Similarly, *next* node denotes the node that appears after the current node u in this ordering. We use `lsa_index` (left `sa_index`) to denote the SA index of the leftmost leaf node that is a descendant of u. Similarly, `rsa_index` (right `sa_index`) denotes the rightmost leaf node that has u as its ancestor. Figure 2 shows an example.

It is simple to determine the `lsa_index` and the leftmost child node of any given node. The properties of the VST and the organization of the VST lead to the following lemma about the VST (we omit the proof for brevity):

Lemma 3. *For a given node in the VST, its rightmost child node, and the right* `sa_index` *can each be determined in constant time.*

3.3 Pattern matching on VST

Lemma 3 provides an indication of how pattern matching can be performed on the VST. For pattern matching using the suffix tree, an important issue is how to quickly locate all the child nodes for a given internal node. In the VST, each node points to its leftmost leaf node using the `sa_index`. During pattern matching, at any given node in the VST, we will need to determine four parameters, namely the leftmost child node (`lchild`), the rightmost child node (`rchild`), the left `sa_index` (`lsa_index`) and the right `sa_index` (`rsa_index`). These parameters define the boundaries of the search at the given node. To search in a leaf node of the VST, we will need only the left

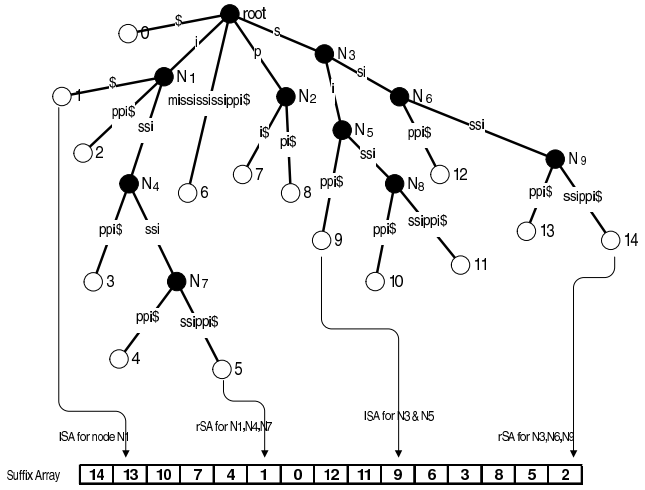


Figure 2. Example VST (solid nodes) showing left SA index (lSA) and right SA index (rSA) for sample nodes.

`sa_index` and right `sa_index` of the node. When we search in an internal node, we will need all the four parameters to match a pattern. Lemma 3 shows that for any given node, we can determine each of these parameters in constant time. The following two examples further illustrate the two cases involved in computing the `rsa_index`, and how pattern matching can be performed on the VST.

Example 4. Determining the right boundary from a next sibling node. Consider node N_5 in Figure 2. The left `sa_index` of N_5 is 9 and the right `sa_index` is 11, since N_5.`sa_index`=9 and N_{5+1}.`sa_index`=12, and hence the right `sa_index` of N_5=12-1=11. The leftmost child node is the `fchild` of the current node, thus the leftmost child of N_5 is N_8. The next node of the rightmost child node is N_{5+1}.`fchild`=N_9. Then the rightmost child node is N_{9-1}=N_8, since the child node will be stored side by side between sibling nodes.

Example 5. Determining the right boundary from the right boundary of the parent node. Consider node N_1 in Figure 2. The left `sa_index` of N_1 is N_1.`sa_index`=1. The right `sa_index` of N_1 is N_2.`sa_index` - (N_1.`nnleaf` -1)=7-1-1=5. The leftmost child node of N_1 is N_1.`fchild`=N_4. The next node of N_1 is N_2. Since N_2.`fchild`=-N_5 is negative, N_2 must be a leaf node in the VST. The right node of N_1 will thus point to N_5. We therefore know that the next node of the rightmost child node of N_1 will be N_5. Finally, the rightmost child node of N_1 can be determined as $N_{5-N_1.\texttt{nnleaf}} = N_{5-1} = N_4$.

We summarize the foregoing discussion as the first main result of this paper:

Theorem 6. *Given a string $T = T[1..n]$ of length n, with symbols from an alphabet Σ, and the virtual suffix tree for T, we can count the number of occurrences of a pattern $P = P[1..m]$ in T in $O(m \log |\Sigma|)$ time, and locate all the η_{occ} occurrences of P in T in $O(m \log |\Sigma| + \eta_{occ})$ time.*

Proof. The theorem is a consequence of Lemma 3. First consider the cost of one single symbol-by-symbol comparison at a node in the VST. The number of child nodes at

any internal node can be no larger than $|\Sigma|$, and we can find the boundaries of the search in constant time. Since the edges are ordered lexically at each internal node, and given the HSAM ordering, matching a single symbol can be done in $O(\log |\Sigma|)$ time steps using binary search. To find the first match, we need to consider the m symbols in the pattern. We perform the above symbol-by-symbol comparisons at most m times to decide whether there is a match or not. After a match is found, we can again use binary search (using `lsa_index` and `rsa_index` as bounds) to determine all the η_{occ} occurrences of the pattern. Reporting each occurrence can be done in constant time, or an additional η_{occ} time for all the occurrences. □

4 Improved Virtual Suffix Tree

The basic data structure introduced above stores the length of each edge in the VST. We can improve the structure to reduce the space requirement by avoiding the need to store information about the edge lengths directly. The improved data structure has only four attributes rather than five. The attributes `sa_index` and `elength` in the basic structure are now combined into one attribute called the adjusted SA index (`asa_index`). This requires a key modification to the suffix tree, leading to an important distinction between the suffix tree and the virtual suffix tree.

4.1 Adjusting edge lengths

A well-known property of the suffix tree is that no two edges out of a node in the tree can start with the same symbol. For efficient representation of the VST, this characteristic of the ST is modified such that, for a given node, every edge that leads to an internal node in the VST has an equal length. This modification is done as follows: Start from the root node and progress towards the leaf nodes in the VST. For a given internal node, say u, adjust the edge label from u to each of its children such that all edges that lead to an internal node will have the same edge length. The major criteria is that, for two sibling internal nodes, their edge labels differ only in the last symbol. If for some edge, say (u, w), the original edge length (or edge label) is longer than the new length, prepend the extraneous part of old $label(u, w)$ to each outgoing edge from w. The edge length for edges that lead to leaf nodes are left unchanged. Then repeat the adjustment at each child node of u. Figure 3 shows an example of this procedure. We can observe that this adjustment only affects the edge lengths, and does not change the general topology of the suffix tree.

The above adjustment procedure leads to an important property of the VST:

Property: *In the improved VST, all internal sibling nodes occur at the same node-depth, and same string-depth, and the edge labels for the edges from the parent to each sibling differ only in the last symbol. This means that, in the VST, two branches from the same node can start with the same symbol, but their edge labels will differ.*

This property provides an important difference between the suffix tree and the VST. The suffix tree mandates that no two edges from the same node have the same starting symbol. Further, the suffix tree only guarantees that the node-depth of two sibling nodes are the same, but not their string depth. This property of equal-length sibling edge labels is the key to more efficient representation of the VST, without explicit edge labels. Figure 4 shows an example of the modified suffix tree with equal-length edges for sibling nodes that are also internal nodes, and the corresponding improved virtual suffix tree. Table 2 shows the corresponding values of the attributes

for each node in the improved VST. What remains is how we compute `asa_index`, the adjusted SA index. This is done by combining the original `sa_index` with `elength`. We state the following lemma without proof:

Lemma 7. *Given a node in the VST say u, and its parent node (say v), we can compute the adjusted SA index in constant time. Further, when required, the edge length can be determined in constant time.*

While we store only the `asa_index`, our calculations will still use the original `sa_index`. However, this can be derived from `asa_index` in constant time. In fact, we can observe that in practice, we need to compute the `asa_index` for only the leftmost child node at each node-level, while keeping the original `sa_index` for all other nodes. To determine the `new_elength` for these other nodes, we simply make a constant time access to their leftmost (sibling) node (at the same node-level), and then use this to compute the length. For searching with the VST, we will calculate the length of the common string at each level. If the length is greater than 0, then we know there is a common string in the child nodes and only the last character is different. Thus, we do not need to store the edge lengths explicitly, leading to a reduction of one integer per node over the basic VST.

NodeName	root	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9
`sa_index`	0	1	7	9	3	9	12	4	10	13
`fchild`	N_1	N_4	-N_5	N_5	N_7	N_8	N_9			
`new_elength`	0	1	1	1	1	1	1	3	1	2
`nfleaf`	1	2	2	0	1	1	1	2	2	2
`nnleaf`	0	1	0	0	0	0	0	0	0	0
`asa_index`	0	1	7	9	3	9	12	4+3=7	10	13+2=15

Table 2. Node attributes in the improved VST for the example sequence, T = `mississisippi$`. We have included `new_elength`, so one can compare with `elength` in Table 1. However, in practice this will not be stored in the VST.

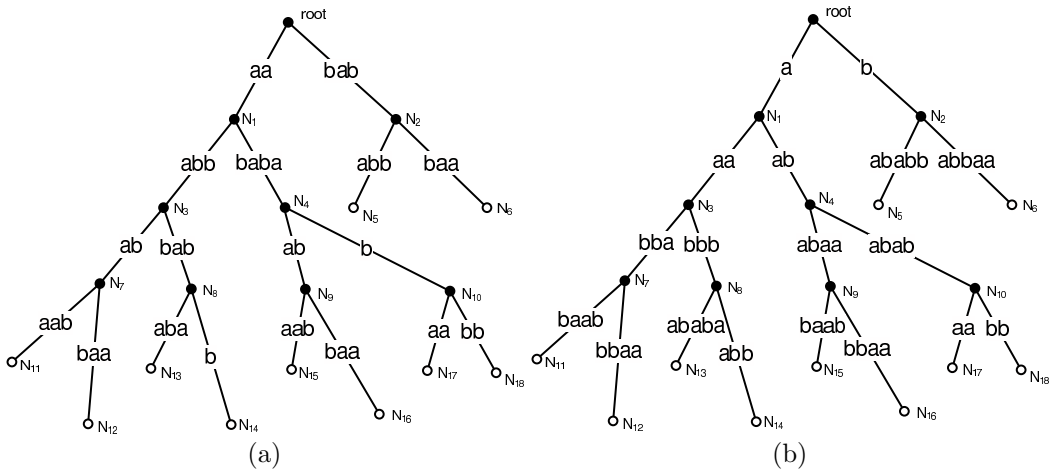


Figure 3. VST edge-length adjustment procedure. (a) original tree; (b) improved tree after adjusting the edge lengths.

4.2 Construction algorithm

Construction of the VST makes use of an array Q which records the internal nodes of the suffix tree. This array maps the internal nodes of the suffix tree to nodes in the VST. Thus, elements in the array are in the same ordering as the corresponding nodes in the VST.

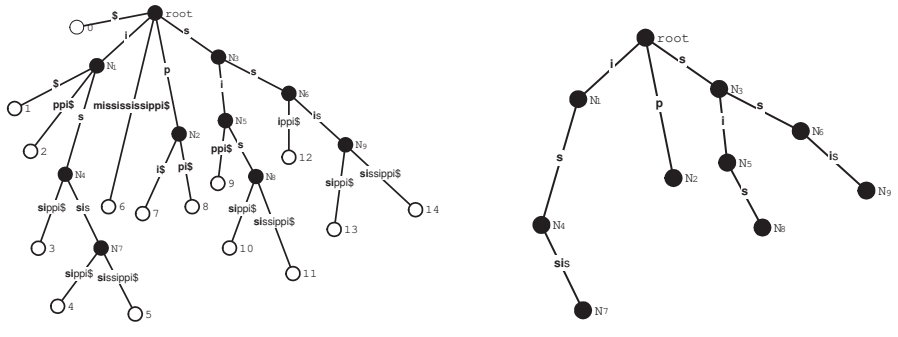
Given an input string T, the first step is to construct the suffix array for T. This can be done in worst case linear time and linear space using any of the existing algorithms [13,18,17,3]. Using the SA, we construct the suffix tree as described in [2]. While the suffix tree can be constructed directly in linear time, working from the SA to the ST will require less space for the construction. The suffix tree is then preprocessed in linear time to adjust the edges from a given parent node that lead to internal child nodes to equal-length edges. Using the adjusted suffix tree, the algorithm will process the internal nodes in the suffix tree in a top-down manner to determine the attributes (`fchild`, `nfleaf` and `nnleaf`) for the corresponding nodes in the VST. Next, we process the VST from the VST leaf nodes to the root, using the Q array to update the `asa_index` at each node. The adjusted `asa_index` field includes information on the `sa_index` and edge length.

The steps for constructing the VST for a given input string are summarized in Algorithm 1.

4.3 Further space reduction

We can further reduce the space needed by the VST, at the cost of an increased time for pattern matching. In the pattern matching phase, if the algorithm is to compare symbols one-by-one, rather than using binary search on the branches from a given node in the VST, we will only need to compute the `lsa_index` and `rsa_index` of the node.

Consider an arbitrary node (say node u) in the VST. The number of children from u or the number of u's leaf nodes cannot be larger than $|\Sigma|$. Thus, the `sa_index` of any child node of u will lie between node u's `lsa_index` and `rsa_index`. Then comparing one symbol from the pattern against the first symbol on each edge from u to its children will require at most $O(|\Sigma|)$ time steps. The left child node and the right child node will not need to be used again. Thus, the attributes `fchild` and



(a) (b)

Figure 4. Improved VST for the string T = `mississississippi$`: (a) modified suffix tree; (b) improved virtual suffix tree

`nfleaf` in the leaf nodes of the VST are no longer required. We make the `asa_index` to be negative for the leaf nodes. Thus, during pattern matching, this serves as a flag for the VST leaf nodes. This compact structure will reduce the space requirement at each leaf node of the VST by 5 bytes. Pattern matching time, however, will increase to $O(|\Sigma|)$ for each symbol in P, or $O(m|\Sigma|)$ overall.

4.4 Complexity Analysis

Time and space complexity The time cost for lines 1-3 in the construction algorithm (Algorithm 1) is $O(n)$+$O(n)$+$O(n)$=$O(n)$. Lines 5-17 in the algorithm perform a one time traversal of the nodes in the suffix tree. The respective values of $pTop$ and $pBottom$ range from 1 to $2n$. Thus the cost for the traversals is $O(n)$. Lines 18-27 in the algorithm run at most $pBottom$ times. The time for lines 18-27 in the algorithm is thus $O(n)$, since each iteration of the loop requires constant time. Therefore, for the regular VST, the overall construction time is $O(n)$. The time for pattern matching is in $O(m \log |\Sigma|)$. For the compact structure, the construction time is the same as the regular structure, but the VST is no longer stored linearly. Here we use an array to store the relation between the Q array and the compact VST. The searching time is now $O(m|\Sigma|)$.

The space requirement clearly depends on the number of nodes in the VST, which is at most n for a sequence of length n. Each node requires a fixed amount of memory to store, leading to an $O(n)$ space requirement.

Number of nodes and practical space requirement The actual space needed for the VST depends on the topology of the suffix tree. This topology can be captured by the number of internal nodes in the suffix tree, or alternatively, by the quantity R_{IL}, the ratio between the number of internal nodes and the number of leaf nodes. We call R_{IL} the *density* or *branching factor* for the suffix tree. We conducted an experiment to evaluate the effect of this branching factor on the storage requirement of the VST. The suffix tree was constructed and the branching factors computed for a set of files taken from [26]. For each file, we used the first 2^{24} symbols as the text, and computed the branching factor. Table 3 shows the results. The maximum ratio of 0.76 was observed for the file `Jdk13c`. On average, however, the maximum ratio was around 0.63. The worst case occurs for a sequence with $|\Sigma| = 1$, (that is, $T = a^n$), leading to a branching factor of 1. The table shows that, for a given sequence, the branching factor depends on a complex relationship between n, $|\Sigma|$, and the mean LCP.

The space requirement for the VST, for both the compact and regular structures depends directly on the branching factor. The last two columns in Table 3 show the maximum space requirement for each file.

The foregoing discussion leads to the following lemma on the construction of the VST:

Lemma 8. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, the virtual suffix tree (without the suffix link) can be constructed in $O(n)$ time, and $O(n)$ space, independent of Σ.*

File	$\|\Sigma\|$	Max Ratio	Compact	Regular	Description
Bible	63	0.61	8.60n	10.13n	King James bible
Chr22	5	0.73	9.50n	11.33n	Human chromosome 22
E.coli	4	0.65	8.89n	10.52n	*Escherichia coli* genome
Etext	146	0.54	8.02n	9.36n	Texts from Gutenberg project
Howto	197	0.55	8.13n	9.51n	Linux Howto files
Jdk13c	113	0.76	9.69n	11.59n	JDK 1.3 documentation
Rctail	93	0.66	8.95n	10.60n	Reuters news in XML format
Rfc	120	0.64	8.77n	10.36n	Concatenated IETF RFC files
Sprot	94	0.61	8.54n	10.05n	
World	94	0.54	8.06n	9.41n	CIA world fact book
Average		0.63	8.71n	10.29n	

Table 3. Branching factor and maximum space requirement for various sample files.

5 Computing Suffix Links

Constructing the suffix tree from the suffix array as described in [2] does not include the suffix link. There are also a number of other suffix tree construction algorithms that build the suffix tree without the suffix link. See Farach et al [8], and Cole and Hariharan [7]. The suffix link, however, is a significant component of the suffix tree, and is important in certain applications, such as approximate pattern matching using matching statistics, and other forms of traversal on the suffix tree. Thus, a data structure to support the complete functionality of the suffix tree requires an inclusion of the suffix link. Recent efficient data structures for suffix trees have thus provided mechanisms for constructing the suffix link. The ESA [1] provided suffix links using complicated RMQ preprocessing [5]. The LST [16] also supported suffix links using the `lcp`-interval tree and intervals defined on the inverse suffix array. A recent work by Maaβ [19] focused exclusively on suffix link construction from suffix arrays, or from suffix trees that do not have such links.

The virtual suffix tree provides a natural mechanism for constructing suffix links. The key idea is that suffix links in the VST can be computed bottom-up, from the nodes with the highest node-depth (leaf nodes) in the VST to those with the least (the root). This is based on the following two observations about suffix links.

1. Consider a leaf node u_{ST} in the suffix tree corresponding to suffix T_i in the original sequence. The suffix link from u_{ST} will point to the leaf node corresponding to the suffix T_{i+1} (that is, the suffix that starts at the next position in the sequence).
2. The suffix link from a node u in the VST will point to some node w with a smaller string-depth in the VST, such that $|L(u)| = |L(w)| + 1$ (or equivalently $|L(u_{ST})| = |L(w_{ST})| + 1$).

The following lemma establishes how we can build suffix links on the VST.

Lemma 9. *Given the VST for a string $T = T[1..n]$ of length n, the suffix links can be constructed in $O(n)$ time using additional $O(n)$ space.*

Proof. Let u and w be two arbitrary nodes in the VST. Let v be the parent node of u. Let $u.\texttt{slink}$ be the node to which the suffix link from node u points to. We consider two cases:

Case A: *u is a leaf node in the VST.* Then, using the above observations, the suffix link from node u will point to node w in the VST (that is, $u.\texttt{slink} = w$) such that

$SA[w.\texttt{sa\_index}] = SA[u.\texttt{sa\_index}] + 1$. Clearly, $|L(w)| = |L(u)| - 1$, where $L(x)$ is the path label of node x. Note that this path label is not explicitly stored in the VST, but for each node, the length can be computed in constant time. This computation can be performed in constant time by maintaining two arrays and observing that $n - |L(w)| = n - |L(u)| + 1$. One array is the inverse suffix array (ISA) for the given string, defined as follows: $ISA[i] = j$ `if` $SA[j] = i$, $(i, j = 1, 2, ..., n)$. The second is an array M that maps the SA values to the corresponding parent nodes in the VST, defined as follows: $M[i] = u$, if u_{ST} in ST is the parent node of the leaf node corresponding to the suffix $T_{SA[i]}$. Clearly, both arrays can be computed in linear time, and require linear space.

Case B: *u is not a leaf node in the VST.* This is a simpler case. When u is an internal node in the VST, the suffix link of u will point to some node w, such that w is an ancestor of node $u.\texttt{fchild.slink}$, such that $|label(u, u.\texttt{fchild})| = |label(w, u.\texttt{fchild.slink})|$. The $O(n)$ time result then follows by using the skip/count trick [11], by observing that a VST has at most n nodes, a node depth of at most n, and that each upward traversal on the suffix link decreases the node depth by at least 1. □

Algorithm 1: VST Construction Algorithm

```
CONSTRUCT-VST(T, n)
 1  SA ← COMPUTE-SUFFIXARRAY(T, n)
 2  ST ← SUFFIXTREE-FROM-SUFFIXARRAY(SA)
 3  ST ← ADJUST-EDGELENGTHS(ST)
 4  Initialize VST[],Q[], pTop=0, pBottom=0, curNode=root, Q[pTop]=root
 5  while (pBottom >= pTop)
 6    for ( each childnode in curNode) do
 7      if (childnode is internal node in ST ) then
 8        pBottom ← pBottom + 1; Q[pBottom] ← childNode
 9       if childnode is first internal node then
10          VST[pTop].fchild ← pBottom
11        end if
12      else
13        Update VST[pTop].nfleaf and VST[pBottom].nnleaf
14      end if
15    end for
16    pTop ← pTop + 1; curNode ← Q[pTop]
17 end while
18 for (pb ← pBottom down to 0) do
19    if (VST[pb] is leaf node) then
20      VST[pb].asa_index ← Q[pb].fchild
21      else if (Q[pb].elength=1) then
22        VST[pb].asa_index←VST[pb].fchild.asa_index
            + VST[pb].nfleaf - Q[pb].elength
23      else
24        VST[pb].asa_index←VST[pb].fchild.asa_index
            + VST[pb].nfleaf - Q[pb].elength+ Q[pb].elength
25      end if
26    end if
27 end for
```

Although the above description is from the viewpoint of a VST already constructed, the suffix links can be constructed as the VST is being constructed, by

some modification of the VST construction algorithm. Algorithm 2 shows a modification of Algorithm 1 (the VST construction algorithm) to incorporate sections to compute the suffix link. The suffix link construction algorithm is based on the Q array used during the VST construction.

Algorithm 2: VST construction with suffix links

```
4    Initialize VST[],Q[],ISA[],M[], pTop←0, pBottom←0, curNode←root, Q[pTop]←root
                ⋮
18   for (pb ← pBottom down to 0) do
19     if (VST[pb] is leaf node) then
20         Update array M to map SA index and node VST [pb]
                ⋮
26     end if
27   end for
28   for (pb ← pBottom down to 0) do
29     if (VST[pb] is leaf node) then
30         VST[pb].slink ← M[ISA[VST[pb].sa_index+1]]
31     else
32         Find ancestor w of VST[pb].fchild.slink s.t.
           |label(w, VST[pb].fchild.slink)|=|label(VST[pb], VST[pb].fchild)|
33         Set VST[pb].slink ← w
34     end if
35   end for
```

Figure 5 shows the result of the suffix link algorithm when applied to the VST of our example string $T =$ `mississississippi$`. Essentially, given the VST, the suffix link is constructed right to left, node-depth by node-depth, starting with the rightmost node at the deepest node-depth, and moving up the VST until we reach the root. Thus, the order of suffix link construction in the example will be $SL_1, SL_2, \ldots, SL_9$.

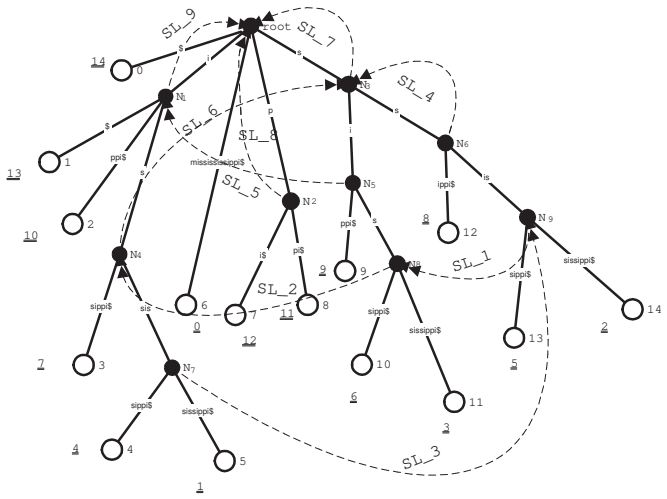


Figure 5. Suffix link on the VST for the sample string $T =$ `mississississippi$`.

Algorithm 2 shows that the additional work required to compute all the suffix links is linear in the length of the string. After construction, the suffix link on the

VST will require one additional integer per internal node in the VST. This can be compared with the 2 integers per node required to store the suffix link using the ESA, or LST. In a typical VST, where the maximum leaf node to internal node ratio is usually less than 0.7, the suffix link will require a maximum total extra space of $0.7n * 4 = 2.8n$ bytes. Table 4 shows the space required for the VST (including the suffix array and suffix links) for both the compact structure and the regular VST, at varying values of the branching factor.

Table 4. Storage requirement for the VST, including suffix links

	Ratio	Compact	Regular
Worst Case	1	15.50n	18.00n
Average Case	0.75	12.63n	14.50n
	0.7	12.05n	13.80n
	0.65	11.48n	13.10n
	0.6	10.90n	12.40n

We summarize the above discussion in the following theorem which captures the second main result of the paper:

Theorem 10. *Given a string $T = T[1..n]$, with symbols from an alphabet Σ, the virtual suffix tree, including the suffix link, can be constructed in $O(n)$ time and $O(n)$ space, independent of Σ.*

Proof. The theorem follows directly from Lemma 8 and Lemma 9. □

6 Conclusion

In this paper, we have presented the virtual suffix tree (VST), an efficient data structure for suffix trees and suffix arrays. The searching performance is the same as the suffix tree, that is, $O(m \log |\Sigma|)$ for a pattern of length m, with symbol alphabet Σ. We also showed how suffix links can be constructed on the VST in linear time, independent of the alphabet size. The VST does not store the edge lengths explicitly. This is achieved by modifying a key property of the suffix tree - the requirement that no two edges from a given node in the suffix tree can start with the same symbol. This key modification leads to a major distinction between the VST and the suffix tree, and results in extra space saving. However, whenever needed, the length for any arbitrary edge in the VST can be obtained in constant time using a simple computation. A further space reduction leads to a more compact representation of the VST, but at the expense of an increased search time, from $O(m \log |\Sigma|)$ to $O(m|\Sigma|)$.

The space requirement depends on the topology of the suffix tree, in particular on the branching factor. For the compact structure, the worst case space requirement (including the suffix array) is $11.5n$ bytes without suffix links, and $15.5n$ bytes with suffix links, where n is the length of the string. However, in practice, the branching factor is typically less than 0.7. For the compact structure, this gives less than $9.25n$ bytes on average without the suffix links, or $12.05n$ bytes with suffix links.

In this work, we have focused on efficient storage of the suffix tree and suffix array after they have been constructed. Thus, we constructed the VST from the suffix tree, which in turn was constructed from the suffix array. An interesting question is whether the virtual suffix tree can be constructed directly, without the intermediate suffix tree stage. This could lead to a significant reduction in space requirement at the time of VST construction.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays.* J. Discrete Algorithms, 2(1) 2004, pp. 53–86.
2. D. Adjeroh, T. Bell, and A. Mukherjee: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays and Pattern Matching*, Springer, to appear, 2008.
3. D. Adjeroh and F. Nan: *Suffix sorting via Shannon-Fano-Elias codes*, in DCC, IEEE Computer Society, 2008, p. to appear.
4. A. Andersson and S. Nilsson: *Efficient implementation of suffix trees.* Softw., Pract. Exper., 25(2) 1995, pp. 129–141.
5. M. A. Bender and M. Farach-Colton: *The LCA problem revisited.*, in LATIN, G. H. Gonnet, D. Panario, and A. Viola, eds., vol. 1776 of Lecture Notes in Computer Science, Springer, 2000, pp. 88–94.
6. M. Burrows and D. J. Wheeler: *A block-sorting lossless data compression algorithm*, Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California, May 1994.
7. R. Cole and R. Hariharan: *Faster suffix tree construction with missing suffix links*, in STOC, 2000, pp. 407–415.
8. M. Farach-Colton, P. Ferragina, and S. Muthukrishnan: *On the sorting-complexity of suffix tree construction.* Journal of the ACM, 47(6) 2000, pp. 987–1011.
9. R. Giegerich, S. Kurtz, and J. Stoye: *Efficient implementation of lazy suffix trees.* Software — Practice and Experience, 33(11) 2003.
10. R. Grossi and J. S. Vitter: *Compressed suffix arrays and suffix trees with applications to text indexing and string matching.* SIAM Journal on Computing, 35(2) 2005, pp. 378–407.
11. D. Gusfield: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, Cambridge University Press, 1997.
12. J. Kärkkäinen: *Suffix cactus: A cross between suffix tree and suffix array*, in CPM: 6th Symposium on Combinatorial Pattern Matching, 1995.
13. J. Kärkkäinen, P. Sanders, and S. Burkhardt: *Linear work suffix array construction.* Journal of the ACM, 53(6) 2006, pp. 918–936.
14. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park: *An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size*, in 12th Annual Symposium on Combinatorial Pattern Matching, 2001.
15. D. K. Kim, J. E. Jeon, and H. Park: *An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size*, in SPIRE 2004, 2004.
16. D. K. Kim, M. Kim, and H. Park: *Linearized suffix tree: an efficient index data structure with the capabilities of suffix trees and suffix arrays.* Algorithmica, 2007.
17. D. K. Kim, J. S. Sim, H. Park, and K. Park: *Constructing suffix arrays in linear time.* J. Discrete Algorithms, 3(2-4) 2005, pp. 126–142.
18. P. Ko and S. Aluru: *Space efficient linear time construction of suffix arrays.* J. Discrete Algorithms, 3(2-4) 2005, pp. 143–156.
19. M. G. Maaβ: *Computing suffix links for suffix trees and arrays.* Information Processing Letters, 101(6) 2007.
20. V. Mäkinen: *Compact suffix array – a space-efficient full-text index.* Fundam. Inform., 56(1-2) 2003, pp. 191–210.
21. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM Journal on Computing, 22(5) 1993, pp. 935–948.
22. E. M. McCreight: *A space-economical suffix tree construction algorithm.* Journal of the ACM, 23(2) 1976, pp. 262–272.
23. K. Monostori, A. Zaslavsky, and H. Schmidt: *Suffix vector: Space- and time-efficient alternative to suffix trees*, in Twenty-Fifth Australasian Computer Science Conference (ACSC2002), M. J. Oudshoorn, ed., Melbourne, Australia, 2002, ACS.
24. J. I. Munro, V. Raman, and S. S. Rao: *Space efficient suffix trees.* J. Algorithms, 39(2) 2001, pp. 205–222.
25. E. Prieur and T. Lecroq: *From suffix trees to suffix vectors*, in Prague Stringology Conference(PCS2005), Prague, 2005.
26. S. J. Puglisi, W. F. Smyth, and A. Turpin: *A taxonomy of suffix array construction algorithms.* ACM Computing Surveys, 39(2) 2007.
27. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
28. P. Weiner: *Linear pattern matching algorithm.* Proceedings, 14th IEEE Symposium on Switching and Automata Theory, 21 1973, pp. 1–11.