PREPRINT. In Proc. Combinatorial Pattern Matching, 1996.

Suffix Trees on Words

Arne Andersson N. Jesper Larsson Kurt Swanson

Dept. of Computer Science, Lund University,
Box 118, S-221 00 LUND, Sweden
{arne,jesper,kurt}@dna.lth.se

Abstract. We discuss an intrinsic generalization of the suffix tree, designed to index a string of length n which has a natural partitioning into m multi-character substrings or *words*. This *word suffix tree* represents only the m suffixes that start at word boundaries. These boundaries are determined by *delimiters*, whose definition depends on the application. Since traditional suffix tree construction algorithms rely heavily on the fact that *all* suffixes are inserted, construction of a word suffix tree is nontrivial, in particular when only $O(m)$ construction space is allowed. We solve this problem, presenting an algorithm with $O(n)$ expected running time. In general, construction cost is $\Omega(n)$ due to the need of scanning the entire input. In applications that require strict node ordering, an additional cost of sorting $O(m')$ characters arises, where m' is the number of distinct words. In either case, this is a significant improvement over previous solutions.
Furthermore, when the alphabet is small, we may assume that the n characters in the input string occupy $o(n)$ machine words. We illustrate that this can allow a word suffix tree to be built in sublinear time.

1 Introduction

The *suffix tree* [14] is a very important and useful data structure for many applications [4]. Traditional suffix tree construction algorithms rely heavily on the fact that *all* suffixes are inserted, in order to obtain efficient time bounds. Little work has been done for the common case where only certain suffixes of the input string are relevant, despite the savings in storage and processing times that are to be expected from only considering these suffixes.

Baeza-Yates and Gonnet [5] have pointed out this possibility, by suggesting inserting only suffixes that start with a word, when the input consists of ordinary text. They imply that the resulting tree can be built in $O(n\mathcal{H}(n))$ time, where $\mathcal{H}(n)$ denotes the height of the tree. Unfortunately, in the worst case, this is quadratic in the size of the input. Also, Giegerich and Kurtz [8] have presented an algorithm that stores certain restricted types of suffixes. Here as well, the worst case time complexity is quadratic, although an advantageous practical behavior has been demonstrated for their specific application.

The aforementioned algorithms have one important advantage compared to traditional suffix tree construction algorithms: they only use $O(m)$ space. Unfortunately, this is obtained at the cost of a greatly increased time complexity. We show that this is an unnecessary tradeoff.

We formalize the concept of words to suit various applications and present a generalization of suffix trees, which we call *word suffix trees*. These trees store, for a string of length n in an arbitrary alphabet, only the m suffixes that start at word boundaries. Linear construction time is maintained, which in general is optimal, due to the requirement of scanning the entire input. This is an asymptotic improvement over Baeza-Yates and Gonnet [5], and Giegerich and Kurtz [8].

Definitions and Main Results

We study the following formal problem: We are given an input string consisting of n characters from an alphabet of size k, including two, possibly implicit, special characters \$ and $\flat$. The \$ character is an end marker which must be the last character of the input string and may not appear elsewhere, while $\flat$ represents some *delimiting character* which appears in $m-1$ places in the input string. We regard the input string as a series of *words*—the m substrings ending either with $\flat$ or \$. There may of course exist multiple occurrences of the same word in the input string. We denote the number of *distinct* words by m'. For convenience, we have chosen to regard each $\flat$ or \$ character as being contained in the preceding word. This implies that there are no empty words; the shortest possible word is a single $\flat$ or \$. The goal is to create a trie structure containing m strings, namely the suffixes of the input string that start at the beginning of words.

Figures 2 and 3 constitute an example where the input consists of a DNA sequence, and the character T is viewed as the word delimiter. (This is a special example, constructed for illustrating the algorithm, *not* a practical case.) Figure 3b shows the word suffix tree for the string shown in Figure 2a. These figures are more completely explained throughout this article.

Our definition can be generalized in a number of ways to suit various practical applications. The $\flat$ character does not necessarily have to be a single character, we can have a *set* of delimiting characters, or even sets of delimiting *strings*, as long as the delimiters are easily recognizable.

Below, we list the main results of this article, which require the following definition:

Definition 1. A *lexicographic* trie is a trie for which the leaves appear in lexicographical order in an in-order traversal. A *non-lexicographic* trie is not guaranteed to have this property.

Theorem 5. *A word suffix tree for an input string of size* n *containing* m *words can be built in* $O(n)$ *expected time and* $O(m)$ *deterministic space.*

Theorem 6. *A lexicographic word suffix tree for an input string of size* n *containing* m *words of which* m' *are distinct can be built in* $O(n + s(m'))$ *expected time and* $O(m)$ *deterministic space, where* $s(m')$ *is the time required to sort* m' *characters.*

Theorem 7. *A lexicographic word suffix tree for an input string of size* n *containing* m *words of which* m' *are distinct can be built deterministically in* $O(n +$

$i(m, m')$) time and $O(m)$ space, where $i(m, m')$ is the time required to insert m characters into ordered dictionaries each bounded in size by m'.

In Section 6 we show that construction of a word suffix tree can even be done in sublinear time in cases where the n characters of the input occupy $o(n)$ machine words.

2 Data Structure

All tries discussed (the word suffix tree as well as some temporary tries) are assumed to be path compressed, i.e., they contain no nodes with out-degree one (other than possibly the root), and the edges of the trie are labeled with strings rather than characters. In order to reduce space requirements, these strings are represented by pointers into the original string. Thus, a trie with m leaves occupies $O(m)$ space.

When choosing a trie implementation, it is important to be aware of which types of queries are expected. One important concept is the ordering of the nodes. Maintaining a lexicographic trie may be useful in some applications, e.g. to facilitate neighbor and range search operations.

Most of the work done on suffix tree construction seems to assume that a suffix tree is implemented as a lexicographic trie. However, it appears that the majority of the applications of suffix trees, for example all those given by Apostolico [4], do not require a lexicographic trie. Indeed, in his classic article on suffix tree construction, McCreight [12] concludes that the use of hash coding, which implies a non-lexicographic trie, appears to be the best representation.

Because of the factors of different applications, it is necessary to discuss several versions of tries. We consider specifically the following possibilities:

1. Each node is implemented as an array of size k. This allows fast searches, but consumes a lot of space for large alphabets.
2. Each node is implemented as a linked list or, preferably, as a binary search tree. This saves space at the price of a higher search cost, when the alphabet is not small enough to be regarded as constant.
3. Hash coding is used within each node (randomized construction). Using dynamic perfect hashing [6], we are guaranteed that searches spend constant time per node, even for a non-constant alphabet. Furthermore, this representation may be combined with variant 2.
4. Instead of storing pointers into the input string for each edge, the pointers are stored only at the leaves, and the character distinguishing each edge is stored explicitly. Thereby, the input string is not accessed while traversing internal nodes during search operations. Hence, an input string stored in secondary memory needs to be accessed only once per search operation.

In the following sections, we assume that the desired data structure is a *non-lexicographic* trie and that a randomized algorithm is satisfactory, except where otherwise stated. This makes it possible to use hash coding to represent trees all through the construction. However, we also discuss the creation of lexicographic suffix trees, as well as non-randomized construction algorithms.

An important fact is that a non-lexicographic trie can be made lexicographic at low cost by sorting the edges according to the first character of each edge, and then rebuilding the tree in the sorted order. We state this in the following observation:

Observation 2. A non-lexicographic trie with l leaves can be made lexicographic in time $O(l + s(l))$, where $s(l)$ is the time required to sort l characters.

3 Wasting Space: Algorithm A

We first observe the possibility of creating a word suffix tree from a traditional (linear size) suffix tree. This is relatively straightforward. Delimiters are not necessary when this method is used—the suffixes to be represented can be chosen arbitrarily. Unfortunately however, it requires much extra space during construction.

We refer to the procedure as *Algorithm A*:

1. Build a traditional *non-lexicographic* suffix tree for the input string in $O(n)$ time with a traditional algorithm [12, 13], using hashing to store edges.
2. Refine the tree into a word suffix tree: remove the leaves that do not correspond to any of the desired suffixes, and compress each pair of nodes connected by a single edge into one node. The time for this is bounded by the number of nodes in the original tree, i.e. $O(n)$.
3. If so desired, make the trie lexicographic in time $O(m + s(m))$ (by Observation 2), where $s(m)$ denotes the time to sort m characters.

If the desired final result is a non-lexicographic tree, the construction time is $O(n)$, the same as for a traditional suffix tree. If a sorted tree is desired however, we have an improved time bound of $O(n + s(m))$ compared to the $\Theta(n + s(n))$ time required to create a lexicographic traditional suffix tree on a string of length n. We state this in the following observation:

Observation 3. A word suffix tree on a string of length n with m words can be created in $O(n)$ time, using $O(n)$ space, and made lexicographic at an extra cost of $O(m + s(m))$, where $s(m)$ denotes the time to sort m characters.

The disadvantage of Algorithm A is that even the most space-economical implementation of Ukkonen's or McCreight's algorithm requires approximately $5n$ values in the range $[0, n]$ to be held in primary storage during construction, in addition to the n characters of the string. While this is infeasable in many cases, it may well be possible to store the final word suffix tree of size $\Theta(m)$.

4 Saving Space: Algorithm B

In this section we present *Algorithm B*, the main word suffix tree construction algorithm, which in contrast to Algorithm A uses only $\Theta(m)$ space.

The algorithm has the following outline: First, a non-lexicographic trie with m' leaves is built, containing all distinct words: the *word trie*. Next, this trie is traversed and each leaf (corresponding to each distinct word in the input string) is assigned its in-order number. Thereafter, the input string is used to create a string of m numbers by representing every word in the input by its in-order number in the word trie. A *lexicographic* suffix tree is constructed for this string. This number-based suffix tree is then expanded into the final non-lexicographic word suffix tree, utilizing the word trie.

Below, we discuss the stages in detail.

1. Building the word trie. We employ a recursive algorithm to create a non-lexicographic trie containing all distinct words. Since the delimiter is included at the end of each word, no word can be a prefix of another. This implies that each word will correspond to a leaf in the word trie. We use hash coding for storing the outgoing edges of each node. The construction is performed top-down by the following algorithm, beginning at the root, which initially contains all words:

1. If the current node contains only one word, return.
2. Set the variable i to 1.
3. Check if all contained words have the same ith character. If so, increment i by one, and repeat this step.
4. Let the incoming edge to the current node be labeled with the substring consisting of the $i-1$ character long common prefix of the words it contains. If the current node is the root, and $i > 1$, create a new, unary, root above it.
5. Store all distinct ith characters in a hash table. Construct children for all distinct ith characters, and split the words, with the first i characters removed, among them.
6. Apply the algorithm recursively to each of the children.

Each character is examined no more than twice, once in step 3 and once in step 5. For each character examined, steps 3 and 5 perform a constant number of operations. Furthermore, steps 2, 4, and 6 take constant time and are performed once per recursive call, which is clearly less than n. Thus, the time for construction is $O(n)$.

2. Assigning in-order numbers. We perform an in-order traversal of the trie, and assign the leaves increasing numbers in the order they are visited, as shown in Figure 2. At each node, we take the order of the children to be the order in which they appear in the hash table. It is crucial for the correctness of the algorithm (Stage 5), that the following property holds:

Definition 4. An assignment of numbers to strings is *semi-lexicographic* if and only if for all strings, α, β, and γ, where α and β have a common prefix that is not also a prefix of γ, the number assigned to γ is either less or greater than both numbers assigned to α and β.

For an illustration of this, consider Figure 2b. The requirement that the word trie is semi-lexicographic ensures that consequtive numbers are assigned to the strings AGAT and AGAAT, since these are the only two strings with the prefix AGA.

The time for this stage is the same as for an in-order traversal of the word trie, which is clearly $O(m')$, where $m' \leq m$ is the number of distinct words.

3. Generating a number string. In this stage, we create a string of length m in the alphabet $\{1, \ldots, m'\}$.

This is done in $O(n)$ time by scanning the original string while traversing the word trie, following edges as the characters are read. Each time a leaf is encountered, its assigned number is output, and the traversal restarts from the root.

4. Constructing the number-based suffix tree. In this stage we create a traditional *lexicographic* suffix tree from the number string. For this, we use an ordinary suffix tree construction algorithm, such as McCreight's [12] or Ukkonen's [13]. Edges are stored in a hash table. The time needed for this is $O(m)$.

Since hash coding is used, the resulting trie is non-lexicographic. However, it follows from Observation 2 that it can be made lexicographic in $O(m)$ time using bucket sorting. In the lexicographic trie, we represent the children at each node with linked lists, so that the right sibling of a node can be accessed in constant time.

5. Expanding the number-based suffix tree. In this stage, each node of the number-based suffix tree is replaced by a local trie, containing the words corresponding to the children of that node. First, we preprocess the word trie for least common ancestor retrieval in $O(m')$ time, using the method of Harel and Tarjan [10]. This allows lowest common ancestors to be obtained in constant time. The local tries are then built left-to-right, using the fact that since the assignment of numbers to words is semi-lexicographic and the number-based suffix tree is lexicographic, each local trie has the essential structure of the word trie with some nodes and edges removed. We find the lowest common ancestor of each pair of adjacent children in the word trie, and this gives us the appropriate insertion point (where the two words diverge) of the next node directly. The node expansion is illustrated in Figure 3a–b.

More specifically, after preprocessing for computation of lowest common ancestors, we build the local trie at each node in the following manner:

1. Insert the first word.

2. Retrieve the next word in left-to-right order from the sorted linked list of children. Compute the lowest common ancestor of this word and the previous word in the word trie.
3. Look into the partially built trie to determine where the lowest common ancestor of the two nodes should be inserted, if it is not already there. This is done by searching up the tree from the last inserted word until reaching a node that has smaller height within the word trie.
4. If necessary, insert the internal (lowest common ancestor) node, and insert the leaf node representing the word.
5. Repeat from step 2 until all children have been processed.

Step 1 takes constant time, and is executed once per internal node of the number-based suffix tree. This makes a total of $O(m')$ time for Step 1. Steps 2, 4 and 5 also take constant time, and are executed once per node in the resulting word suffix tree. This implies that their total cost is $O(m)$. The total work performed in Step 3 is essentially an in-order traversal of the local subtree being built. Thus, the total time for Step 3 is proportional to the total size of the final tree, which is $O(m)$. Consequently, the expansion takes a total of $O(m)$ time.

The correctness of the algorithm can be verified in a straightforward manner. The crucial point is that the number-based suffix tree has the essential structure of the final word suffix tree, and that the expansion stage does not change this.

Theorem 5. *A word suffix tree for an input string of size n containing m words can be built in $O(n)$ expected time and $O(m)$ deterministic space.*

5 Extensions and Variations

Although the use of non-lexicographic suffix trees and randomization (hash coding) during construction is sufficient for a majority of practical applications, we describe extensions to Algorithm B in order to meet stronger requirements.

5.1 Building a Lexicographic Trie

If the alphabet size k is small enough to be regarded as a constant, it is trivial to modify Algorithm B to create a lexicographic tree in linear time: instead of hash tables, use an array of size k to store the children in each node.

If hashing is used during construction as described in the previous section, Algorithm B can be modified to construct a lexicographic trie simply by requiring the number assignments in Stage 2 to be lexicographic instead of semi-lexicographic. Thereby, the number assignment reflects the lexicographic order of the words exactly, and this order propagates to the final word suffix tree. A lexicographic number assignment can be achieved by ensuring that the word trie constructed in Stage 1 is lexicographic. Observation 2 states that the trie can be made lexicographic at an extra cost which is asymptotically the same as for sorting m' characters, which yields the following:

Theorem 6. *A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built in $O(n + s(m'))$ expected time and $O(m)$ deterministic space, where $s(m')$ is the time required to sort m' characters.*

5.2 A Deterministic Algorithm

A non-randomized version of Algorithm B can be obtained by representing the tree with only deterministic data structures, such as binary search trees. Also, as a side effect, the resulting tree becomes lexicographic. This yields a better worst case time, at the price of an asymptotically inferior expected performance.

We define $i(m, m')$ to denote the time to insert m characters into ordered dictionaries each bounded in size by m', where $m' \leq m$ is the number of distinct words. In a straightforward manner, we can replace the hash tables of Stages 1 and 4 with deterministic data structures. Since no node may have higher outdegree than m', the resulting time complexity is $O(n + i(m, m'))$.

Theorem 7. *A lexicographic word suffix tree for an input string of size n containing m words of which m' are distinct can be built deterministically in $O(n + i(m, m'))$ time and $O(m)$ space, where $i(m, m')$ is the time required to insert m characters into ordered dictionaries each bounded in size by m'.*

Using binary search trees, we have $i(m, m') = O(m \log m')$. There are other possibilities, for example we could implement each node as a fusion tree [7], which implies $i(m, m') = O(m \frac{\log m'}{\log \log m'})$.

6 Sublinear Construction: Algorithm C

In some cases, particularly when the alphabet is small, we may assume that the n characters in the input string occupy $o(n)$ machine words. Then it may be possible to avoid the apparently inescapable $\Omega(n)$ cost due to reading the input. This theme can be altered in many ways, the details depend on the application. We start by studying the case when the positions of the delimiters are known in advance. Then we discuss an application where the input string can be scanned and delimiters located in $o(n)$ time.

If the alphabet size is k, then each character occupies $\log k$ bits and the total length of the input is $N = n \log k$ bits stored in N/w machine words, where w is the number of bits in a machine word. We note the following:

Observation 8. A lexicographic trie containing strings of a-bit characters can be transformed into the corresponding lexicographic trie in a b-bit alphabet in linear time, where a and b are integers no larger than the size of a machine word.

The following algorithm, which we refer to as *Algorithm C*, builds a word suffix tree, while temporarily viewing the string as consisting of n' b-bit pseudo-characters, where $n' = o(n)$. It is necessary that this transformation does not cause the words to consist of fractions of pseudo-characters. Therefore, in the case where a word ends at the ith bit of a pseudo-character, we pad this word implicitly with $b - i$ bits at the end, so that the beginning of the next word may start with an unbroken pseudo-character. This does not influence the structure of the input string, since each distinct word can only be replaced by another distinct word. Padding may add at most $m(b-1)$ bits to the input. Consequently,

$$n' = O\left(\frac{N + m(b-1)}{b}\right) = O\left(\frac{N}{b} + m\right).$$

We are now ready to present Algorithm C:

1. Construct a non-lexicographic word trie in the b-bit alphabet in time $O(n')$, as in Stage 1 of Algorithm B. The padding of words does not change the important property of direct correspondence between the words and the leaves of the word trie.
2. Sort the edges of this trie, yielding a lexicographic trie in the b-bit alphabet in $O(m' + s_b(m'))$ time, by Observation 2, where $s_b(m')$ is the time to sort m' b-bit integers.
3. Assign the leaves in-order numbers, and generate the number string in time $O(n')$, analogously to Stage 3 of Algorithm B.
4. Convert this word trie into a word trie in the original k-size alphabet, utilizing Observation 8. (This does not affect the in-order numbers of the leaves).
5. Proceed from Stage 4 of Algorithm B.

The first four steps take time $O(n' + s_b(m'))$, and the time for the completion of the construction from Stage 4 is $O(m)$. Thus the complexity of Algorithm C is $O(n' + m + s_b(m'))$. Thereby we obtain the following theorem:

Theorem 9. *When the positions of all delimiters are known, a lexicographic word suffix tree on a string of n characters comprising m words of which m' are distinct, can be constructed in time*

$$O\left(\frac{N}{b} + m + s_b(m')\right)$$

for some integer parameter $b \leq w$, where N is the number of bits in the input and $s_b(m')$ is the time to sort m' b-bit integers.

Note that Theorem 9 does not give a complete solution to the problem of creating a word suffix tree. We still have to find the delimiters in the input string, which may take linear time. Below, we illustrate a possible way around this for one application.

Example: Huffman Coded Text Suppose we are presented with a Huffman coded text and asked to generate an index on every suffix starting with a word. Furthermore, suppose that word boundaries are defined to be present at every position where a non-alphabetic character (a space, comma, punctuation etc.) is followed by an alphabetic character (a letter), i.e., we have *implicit* $\flat$ characters in these positions. The resulting word suffix tree may be a binary trie based on the Huffman codes, or a trie based on the original alphabet. Here we assume the former.

For simplicity, we assume that the length of the longest Huffman code is less than $\frac{\log n}{4}$. We view the input as consisting of b-bit pseudo-characters, where $b = \frac{\log n}{2}$. We start by creating a table containing 2^b entries, each entry corresponding to one possible pseudo-character. For each entry, we scan the corresponding pseudo-character and examine its contents by decoding the Huffman codes contained in it. If there is an incomplete Huffman code at the end, we make a note to the length of this code. We denote the decodable part of the pseudo-character as a *chunk*. Since the length of the longest Huffman code is less than $b/2$, the length of each chunk is more than $b/2$ bits. While decoding the contents of the chunk, we check if any word boundaries are contained. If so, this is noted in the table. Furthermore, we check if the last character is non-alphabetic, in which case we note that this character, together with the first character in the next chunk, may define a word boundary. The time to examine the table is at most proportional to the total number of bits, which is $2^b \cdot b$.

Now we can read the text and determine the delimiter positions, using one read operation per chunk (or possibly two, if a chunk spans over two machine words). Denoting the number of bits in the input by N, the cost of reading is

$$O\left(\frac{N}{b} + m\right)$$

The total cost for finding delimiters becomes

$$O\left(2^b \cdot b + \frac{N}{b} + m\right) = O\left(\sqrt{n}\log n + \frac{N}{\log n} + m\right) = O\left(\frac{N}{\log n} + m\right),$$

since $N \geq n$. Next, applying Theorem 9 with the same choice of b, we find that $s_b(m') = O(m' + 2^b) = O(m' + \sqrt{n})$ by using bucket sorting. The space used by this algorithm is $O(m + \sqrt{n})$, the last term is due to the table. This yields

Observation 10. Given a Huffman coded input string of length n coded in N bits where no single Huffman code is longer than $\frac{\log n}{4}$ bits, a word suffix tree on m natural words can be constructed in time

$$O\left(\frac{N}{\log n} + m\right),$$

with construction space $O(m + \sqrt{n})$.

7 A Comment on the Time–Space Tradeoff in Practice

Asymptotically, our space requirement is better than that of alternative data structures, like the traditional suffix tree and the suffix array [11] or PAT array [9]. In practice, however, an asymptotic advantage may sometimes be neutralized by high constant factors.

There are two aspects of space efficiency: the space required to construct the data structure and the space required by the data structure after construction.

First, we study the construction space. Recall that we have n characters, m words, and m' distinct words. The space taken by our construction algorithm equals the space required to construct a traditional suffix tree of m characters, plus the space required to store m' words in the word trie, (including lowest common ancestor links). In many practical cases, see Figure 1 for example, m' is considerably smaller than m and we can neglect the space required by the word trie.

Next, we discuss the space required by the final data structure. As a very natural and space-efficient alternative to a tree structure, one may store the suffix references in a plain sorted array, denoted suffix array [11] or PAT array [9]. Such an array may be searched through binary search, but in order to speed up the search, Manber and Myers suggest using an additional bucket array [11]. The idea is to distribute the suffixes among b buckets according to their first $\log b$ bits, as in traditional bucket sorting. Manber and Myers suggested $b = n/4$ as a suitable value, but other values are possible. This method is very helpful is cases when the suffix array is stored in secondary memory; by storing a small bucket array in internal memory, we may decrease the number of disk accesses significantly. One disadvantage with the bucket array, however, is that the number of suffixes may vary a lot between the buckets, many buckets may even be empty. Therefore, Andersson and Nilsson suggested an improvement [3]: instead of the bucket array, a small, efficiently implemented suffix tree may be used to index the array of pointers. As demonstrated through experiment [3], this data structure uses less space than the bucket array while the number of disk accesses is smaller. The reason for this advantage is that a trie in general, and a level compressed trie in particular [1, 2], adapts more nicely to the input distribution than a bucket array. For the same reasons, we conjecture that an efficiently implemented word suffix tree will offer a better time–space tradeoff than a bucket array.

Of course, the main advantage compared with the suffix array is the construction cost; we need only $\Theta(n)$ time (in some cases we also need to sort m' characters) while Manber and Myers' suffix array algorithm requires $\Theta(n \log n)$ time.

8 Comments

The word suffix tree is indeed a natural data structure, and it is surprising that no efficient construction algorithm has previously been presented, although attempts have been made [5, 8]. We now discuss several practical cases where word suffix trees would be desirable.

With natural languages, a reasonable word partitioning would consist of standard text delimiters: space, comma, carriage return, etc. We could also use implicit delimiters, as in the example in the preceding section. Using word suffix trees, large texts can be manipulated with a greatly reduced space requirement, as well as increased processing speed [5]. Figure 1 indicates that the number of words, m, in common novels, is much less than the length of the work in bytes, n. This difference is even greater when one considers the number of distinct words, m'.

Work	n	m	m'
Mark Twain's *Tom Sawyer*	387922	71457	7389
August Strindberg's *Röda rummet*	539473	91771	13425

Fig. 1. Examples of natural language text

In the study of DNA sequences, we may represent a large variety of genetic substructures as words, from representations of single amino acids, up to entire gene sequences. In many such cases, the size of the overlying DNA string is substantially greater than the number of substructures it contains. As an example, there are merely tens of thousands of human genes, whilst the entire length of human DNA contains approximately three billion nucleotides.

Giegerich and Kurtz [8] assert that word suffix trees are very useful in parsing computer languages. This is a very good example of the need to merely store entire words, and not every suffix, as only the language's keywords and variables are relevant search entities.

The word suffix tree is of particular importance when the indexed string is not held in primary storage while the tree is utilized. Using the representation of case 4 in Section 2 our data structure allows search operations with a single access to secondary storage, using only $O(m)$ cells of primary storage, regardless of the length of the search string.

It should be noted that one open problem remains, namely that of removing the use of delimiters, presenting an algorithm that constructs a trie of arbitrarily selected suffixes using only $O(m)$ construction space.

References

1. A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Inf. Process. Lett.*, 46:295–300, 1993.
2. A. Andersson and S. Nilsson. Faster searching in tries and quadtrees—an analysis of level compression. In *Proc. 2^{nd} Annual European Symposium on Algorithms*, pages 82–93. Springer Verlag, 1994.

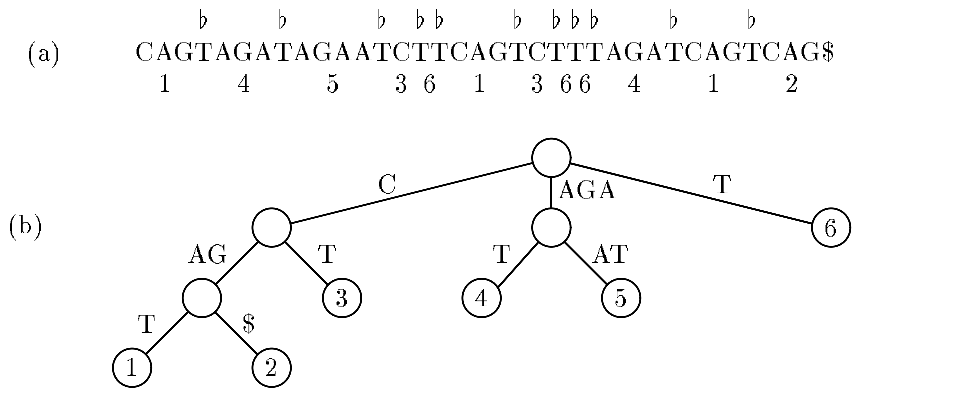

Fig. 2. A sample string where ♭ = T, and its number string (a), created from its corresponding word trie (b).

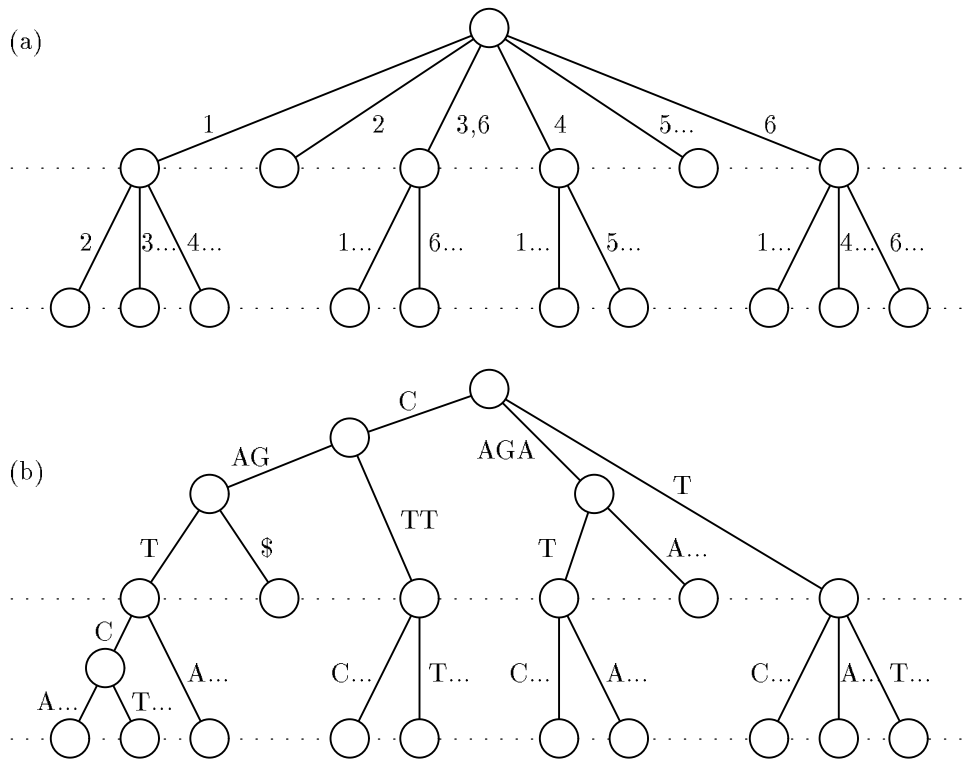

Fig. 3. The number suffix tree (a) and its expansion into the final word suffix tree (b). (Dotted lines denote corresponding levels.)

3. A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software—Practice and Experience*, 25(2):129–141, 1995.
4. A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, NATO ISI Series*, pages 85–96. Springer-Verlag, 1985.
5. R. Baeza-Yates and G. H. Gonnet. Efficient text searching of regular expressions. In *Proceedings of the* 16^{th} *International Colloquium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1989.
6. M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994.
7. M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47:424–436, 1993.
8. R. Giegerich and S. Kurtz. Suffix trees in the functional programming paradigm. In *European Symposium on Programming (ESOP '94)*, volume 788 of *Lecture Notes in Computer Science*, pages 225–240. Springer-Verlag, 1994.
9. G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991. ISBN 0-201-41607-7.
10. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13:338–355, 1984.
11. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, Oct. 1993.
12. E. M. McCreight. A space–economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.
13. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, Sept. 1995.
14. P. Weiner. Linear pattern matching algorithms. In *Proceedings* 14^{th} *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.

This article was processed using the LaTeX macro package with LLNCS style