

On-Line Linear-Time Construction of Word Suffix Trees

Shunsuke Inenaga^{1,2} and Masayuki Takeda^{2,3}

¹ Japan Society for the Promotion of Science

² Department of Informatics, Kyushu University, Fukuoka 812-8581, Japan
{shunsuke.inenaga, takeda}@i.kyushu-u.ac.jp

³ SORST, Japan Science and Technology Agency (JST)

Abstract. Suffix trees are the key data structure for text string matching, and are used in wide application areas such as bioinformatics and data compression. Sparse suffix trees are kind of suffix trees that represent only a subset of suffixes of the input string. In this paper we study *word suffix trees*, which are one variation of sparse suffix trees. Let D be a dictionary of words and w be a string in D^+ , namely, w is a sequence $w_1 \cdots w_k$ of k words in D . The word suffix tree of w w.r.t. D is a path-compressed trie that represents only the k suffixes in the form of $w_i \cdots w_k$. A typical example of its application is word- and phrase-level search on natural language documents. Andersson et al. proposed an algorithm to build word suffix trees in $O(n)$ expected time with $O(k)$ space. In this paper we present a new word suffix tree construction algorithm with $O(n)$ running time and $O(k)$ space in the worst cases. Our algorithm is on-line, which means that it can sequentially process the characters in the input, each by each, from left to right.

1 Introduction

Suffix trees have played a very central role in combinatorial pattern matching as they enable us to solve a multitude of important problems efficiently [3, 8]. To give some examples of applications, suffix trees are utilized in data compression [13, 16, 10] and in bioinformatics such as motif finding [14], regulatory elements discovery [5], and fast protein classification [7]. Suffix trees are fairly useful since they can be constructed in linear time and space with respect to the input string length [19, 15, 18].

On the other hand, there have been great demands to deal with a common case where only certain suffixes of the input string are relevant. Suffix trees that contain only a subset of all suffixes are called *sparse suffix trees*.

The ‘sparsity’ of the suffix tree varies with the application: In [12] Kärkkäinen and Ukkonen proposed the *evenly spaced sparse suffix tree* which contains every i -th suffix for some fixed positive integer i . Their contribution is an algorithm which allows the original full text to be searched, by using the evenly spaced sparse suffix tree. Clifford and Sergot [6] introduced *distributed suffix trees* whose idea is to partition the original suffix tree into a constant number of subtrees and construct each of them in linear time, in parallel. Their suffix tree is thus

helpful to index huge genome sequence databases. Also, sparse suffix trees for a set of arbitrary suffixes are used in the core of pattern discovery algorithms from biological sequences [11, 9].

Another type of sparse suffix trees is *word suffix trees* [4]. Let D be a dictionary of words and w be a string in D^+ , namely, w is a sequence $w_1 \cdots w_k$ of k words in D . The word suffix tree of w w.r.t. D is a tree structure which represents only the k suffixes in the form of $w_i \cdots w_k$. One typical application of word suffix trees is a word- and phrase-level index for documents written in a natural language. Note that normal suffix trees report *any* occurrences of a keyword in the text string, which may cause unwanted matchings (e.g., an occurrence of “other” in “mother” is possibly retrieved).

This paper investigates word suffix tree construction. The most intuitive solution is to build a normal suffix tree using $O(n)$ time and space, then to prune it so that only the leaves corresponding to the k suffixes remain. However, this approach apparently wastes extra space, as the size of the resulting tree is only $O(k)$. To index large text strings efficiently, we need to handle a restricted situation where only $O(k)$ computational space is available. Still, this is a rather challenging task, as traditional linear suffix tree construction algorithms heavily rely on the fact that *all* suffixes are to be inserted in the tree. On the other hand, it is no more true for word suffix trees.

In [2] Andersson et al. took a first step in this problem: they presented an algorithm to build word suffix trees with $O(k)$ working space in $O(n)$ *expected* running time. This present paper takes a further step and puts a period to this problem - our algorithm constructs word suffix trees with $O(k)$ working space in $O(n)$ running time *in the worst cases*. Remark that this is optimal, since the resulting tree requires $O(k)$ space, and we have to read the whole input string at least once and it takes $O(n)$ time. Our algorithm is based on, and is a generalization of, Ukkonen’s on-line suffix tree construction algorithm introduced in [18]. In addition, our algorithm can be seen as a practical solution to efficient construction of general sparse suffix trees.

The rest of the paper is organized as follows. In Section 2 we introduce some definitions and notations. In Section 3 we define word suffix tries and propose an on-line construction algorithm for them. Section 4 presents a word suffix tree construction algorithm, which is the main subject of this paper. Finally, conclusions and further discussions are given in Section 5.

2 Preliminaries

Let Σ be a finite set of symbols, called an *alphabet*. A finite sequence of symbols is called a *string*. We denote the length of a string u by $|u|$. The empty string is denoted by ε , that is, $|\varepsilon| = 0$. Let Σ^* be the set of strings over Σ , and let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. Strings x , y , and z are said to be a *prefix*, *substring*, and *suffix* of the string $u = xyz$, respectively. A prefix, substring, and suffix of a string u are said to be *proper* if they are not u . Let $Prefix(u)$ and $Suffix(u)$ be the set of prefixes and suffixes of string u , respectively. Let $Prefix(S) = \bigcup_{u \in S} Prefix(u)$ for

a set S of strings. The i -th symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the substring of a string u that begins at position i and ends at position j is denoted by $u[i..j]$ for $1 \leq i \leq j \leq |u|$.

Definition 1 (Prefix property). *A set L of strings is said to have the prefix property if no string in L is a proper prefix of another string in L .*

Let $D = \Sigma^* \cdot \#$. Then D is a set of strings each followed by $\#$, and D is called a *dictionary*. We assume that any string w is an element of D^+ . This is a very natural assumption, since for example in the European languages the blank character can be regarded as the special character $\#$, and any text is an element of D^+ .

A *factorization* of string $w \in D^+$ w.r.t. D is a list w_1, \dots, w_k of strings in D such that $w = w_1 \cdots w_k$. Note that this factorization is always unique, since $D = \Sigma^* \cdot \#$ clearly satisfies the prefix property because of $\#$ not being in Σ . Now, let $Suffix_D(w) = \{w_i \cdots w_k \mid 1 \leq i \leq k+1\}$. Remark that $Suffix_D(w)$ is a subset of $Suffix(w)$ which consists only of the original string w and the suffixes which immediately follow $\#$ in w (including the empty suffix ε intended by $w_{k+1}w_k$).

3 Word Suffix Trie

In this section, we present our word suffix *trie* construction algorithm which will be a basis of our word suffix *tree* construction algorithm to be given later as the main topic of this paper.

3.1 Definition

Definition 2 (Word suffix trie). *The word suffix trie of a string $w \in D^+$ w.r.t. D , denoted by $WSTrie_D(w)$, is a trie which represents $Suffix_D(w)$.*

Fig. 1 compares the normal suffix trie and the word suffix trie for string w , where $\Sigma = \{\mathbf{a}, \mathbf{b}\}$, $D = \Sigma^* \cdot \#$, and $w = \mathbf{ab}\#\mathbf{ab}\#\mathbf{a}\#$.

It is easy to see that there is a natural one-to-one correspondence between the nodes of $WSTrie_D(w)$ and the strings in $Prefix(Suffix_D(w))$. Any string u in $Prefix(Suffix_D(w))$ can be written as $u = xy$ such that $x \in D^*$ and y is a *proper* prefix of some string in D . It should be stated that the choice of x and y is unique for each u . Hereafter, we represent a node of $WSTrie_D(w)$ with an ordered pair $\langle x, y \rangle$, as mentioned above.

3.2 Word Suffix Trie Construction Algorithm

Suffix Link. Ukkonen [18] used suffix links for on-line construction of normal suffix tries. Here we give a new definition of suffix links that is suitable for on-line word suffix trie construction.

For dictionary $D = \Sigma^* \cdot \#$, we consider the smallest DFA M_D which accepts D . Clearly it has a unique final state with no outgoing edges (see the left of

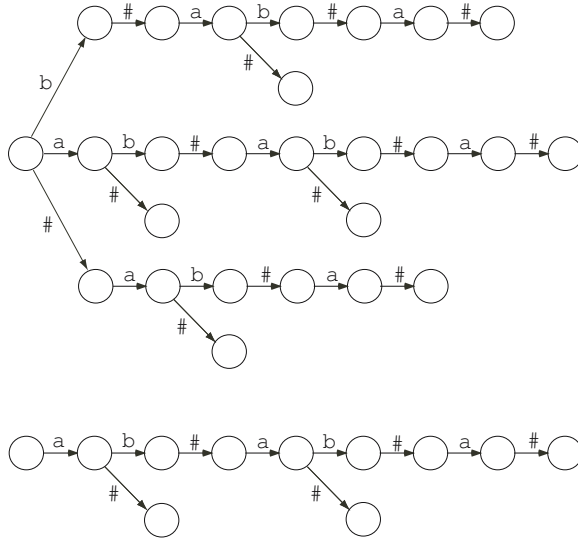


Fig. 1. The normal suffix trie of $w = ab\#ab\#a\#$ on the upper, and the word suffix trie of w w.r.t. $D = \{a, b\}^* \cdot \#$ on the lower. Note that the normal suffix tree represents all the suffixes of w , while the word suffix tree represents only the suffixes $ab\#ab\#a\#, ab\#a\#, a\#, \varepsilon \in Suffix_D(w)$.

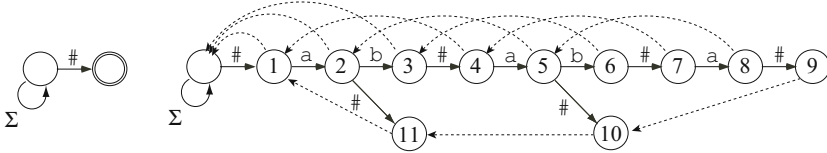


Fig. 2. To the left is the smallest DFA M_D accepting $D = \{a, b\}^* \cdot \#$, and to the right is $WSTrie_D(w)$ for $w = ab\#ab\#a\#$, with M_D and its suffix links (broken arrows) attached. Nodes 4, 5, 6, 7, 8, 9, 10, and 11 are those in Group 1 of Definition 3, and nodes 1, 2, and 3 are those in Group 2.

Fig. 2). Then we attach M_D to the word suffix trie, replacing the unique final state of M_D by the root of the word suffix trie. Now we define the suffix links of word suffix tries as follows:

Definition 3 (Suffix links of word suffix trie). Let $D = \Sigma^* \cdot \#$ and M_D be the smallest DFA that accepts D . For each node $s = \langle x, y \rangle$ of $WSTrie_D(w)$,

1. if $x \in D^+$, the suffix link from s goes to node $\langle x', y \rangle$ such that $x' \in D^*$ and $x = hx'$ for some $h \in D$;
2. otherwise (if $x = \varepsilon$), the suffix link from s goes to the initial state of M_D .

Fig. 2 shows the smallest DFA M_D which accepts $D = \{a, b\}^* \cdot \#$, and $WSTrie_D(w)$ for $w = ab\#ab\#a\#$ with its suffix links.

Algorithm. Fig. 3 shows a pseudo code of our on-line algorithm to build word suffix tries, with the help of DFA M_D and suffix links of Definition 3. Observe that procedure *Update* is identical to that of Ukkonen's on-line normal suffix trie construction algorithm of [18]. The only change is the initialization steps of the main routine where we set the root of the trie to the final state of M_D and the suffix link of the root to the initial state of M_D . This simple modifications make a difference in the resulting data structures. A snapshot of on-line construction of $WSTrie_D(w)$ with the running example is shown in Fig. 4.

<pre> Input: $w = w[1..n] \in D^+$ and auxiliary DFA M_D. Output: Word suffix trie of w w.r.t. D. { $root =$ the final state of M_D; $slink(root) =$ the initial state of M_D; $top = root$; for ($i = 1$; $i \leq n$; $i++$) $top = Update(top, w[i])$; } node $Update(top, c)$ { $newtop = CreateNewNode()$; create a new edge $top \xrightarrow{c} newtop$; $prev = newtop$; for ($t = slink(top)$; no c-edge from t; $t = slink(t)$) { $new = CreateNewNode()$; create a new edge $t \xrightarrow{c} new$; $slink(prev) = new$; $prev = new$; } $slink(prev) =$ the initial node of the c-edge from t; return $newtop$; } </pre>

Fig. 3. Word suffix trie construction algorithm. For any node v , $slink(v)$ represents the node to which the suffix link of v goes. Remark that function *Update* is identical to that of Ukkonen's normal suffix trie construction algorithm [18]. The initialization step using the auxiliary DFA M_D changes the algorithm so that it builds word suffix tries.

For the correctness of the algorithm of Fig. 3, it suffices to show the following lemma:

Lemma 1. *Let $w \in D^+$, w_1, \dots, w_k be a unique factorization of w w.r.t. D . Let j be an integer with $0 \leq j \leq |w|$, and u be the prefix of length j of w . Let $u = w_1 \cdots w_\ell v$ such that v is a proper prefix of $w_{\ell+1}$. After the j -th call of the *Update* operation, we have a trie representing the strings*

$$\{w_i \cdots w_\ell \mid 1 \leq i \leq \ell + 1\} \cdot v.$$

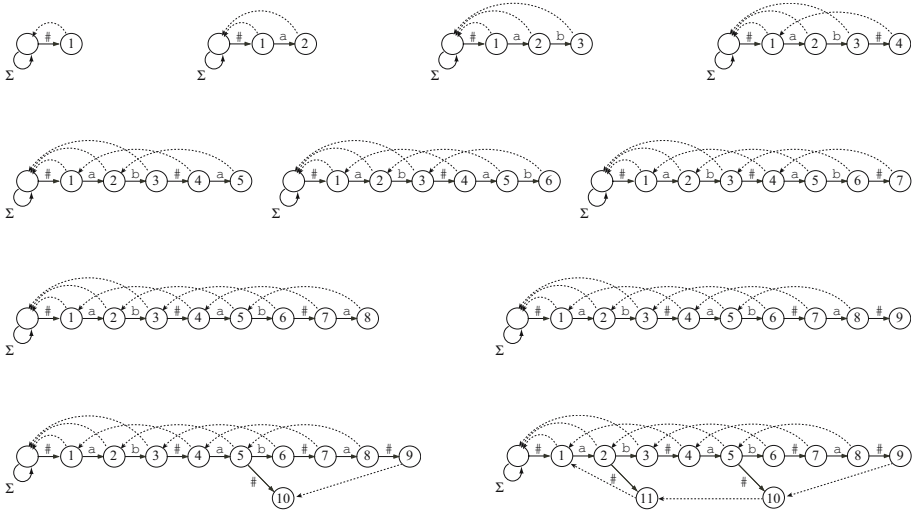


Fig. 4. A snapshot of on-line construction of $WSTrie_D(w)$ with $w = ab\#ab\#a\#$ and $D = \{a, b\} \cdot \#$. The update with the last $\#$ is shown in three steps, where we get three new nodes and edges.

The suffix link of the node $\langle w_i \dots w_\ell, v \rangle$ goes to the node $\langle w_{i+1} \dots w_\ell, v \rangle$, if $i \leq \ell$; and otherwise, goes to the state $\delta(q_0, v)$ of M_D , where δ and q_0 are, respectively, the state-transition function and the initial state of M_D .

Proof. By induction on $j = |u|$. When $|u| = 0$, the lemma trivially holds. We now consider $|u| > 0$. When $v \neq \varepsilon$, let $v = v'b$ with $v' \in \Sigma^*$ and $b \in \Sigma$. By the induction hypothesis, after the $(j - 1)$ -th call of *Update*, we have a trie representing

$$\{w_i \dots w_\ell \mid 1 \leq i \leq \ell + 1\} \cdot v',$$

and the suffix link of node $\langle w_i \dots w_\ell, v' \rangle$ goes to node $\langle w_{i+1} \dots w_\ell, v' \rangle$, if $i \leq \ell$; and otherwise, goes to the state $\delta(q_0, v')$ of M_D . At the j -th call, the variable *top* is set to the node $\langle w_1 \dots w_\ell, v' \rangle$ and the node $\langle w_1 \dots w_\ell, v'b \rangle$ is created (variable *newtop*). In the iteration of the **for** loop, we traverse the suffix links starting at the node $\langle w_1 \dots w_\ell, v' \rangle$. For each $i = 2, \dots, \ell$, the node $\langle w_i \dots w_\ell, v'b \rangle$ is created, if it does not exist. Note that the iteration is guaranteed to halt since the suffix links lead us to the state $\delta(q_0, v')$. During the iteration, the suffix links of the newly created nodes $\langle w_i \dots w_\ell, v'b \rangle$ are set to the nodes $\langle w_{i+1} \dots w_\ell, v'b \rangle$, if $i \leq \ell$; and to the state $\delta(q_0, v'b)$, otherwise. Thus the lemma holds for the case $v \neq \varepsilon$. Similarly, we can prove the case $v = \varepsilon$. □

Remark 1. Our word suffix trie construction algorithm of Fig. 3 generalizes Ukkonen’s normal suffix trie construction algorithm [18]. Assume just for now $D = \Sigma$, and consider a DFA which accepts Σ with only two states that are a single initial state and a single final state. Then this DFA plays the same role as the auxiliary ‘ \perp ’ node used in Ukkonen’s algorithm, and thus our algorithm

builds normal suffix tries. The same discussion applies to the word suffix tree construction algorithm to be given in the next section.

4 Word Suffix Tree

In the previous section, we presented our on-line algorithm that constructs word suffix tries. The drawback is, however, that the size of a word suffix trie can be quadratic in the input string length. In this section, we consider the *word suffix tree* whose size is bounded by $O(k)$, where k is the number of words in string w w.r.t. dictionary D . We then propose a new algorithm to build a word suffix tree in $O(n)$ time with $O(k)$ space, where $n = |w|$ and $w = w_1 \cdots w_k$. The advantage of our algorithm to the one by Andersson et al. [2] is that our algorithm runs in $O(n)$ time *in the worst cases*, while their algorithm runs in $O(n)$ time *on the average*.

4.1 Definitions

Definition 4 (Word suffix tree). *The word suffix tree of a string $w \in D^+$ w.r.t. D , denoted by $WSTree_D(w)$, is a path-compressed trie which represents $Suffix_D(w)$.*

For any strings x, y , let $lcp(x, y)$ denote the longest common prefix of x and y . Let

$$I = \{lcp(w_i \cdots w_k, w_j \cdots w_k) \mid 1 \leq i \neq j \leq k + 1\} \text{ and,}$$

$$E = \{w_i \cdots w_k \mid w_i \cdots w_k \notin Prefix(w_j \cdots w_k) \text{ for any } 1 \leq j < i \leq k\}.$$

Then, there is a one-to-one correspondence between the strings in I and the internal nodes (including the root) of $WSTree_D(w)$, and there is a one-to-one correspondence between the strings in E and the leaves of $WSTree_D(w)$. Hereafter, we sometimes refer to any node s of $WSTree_D(w)$ as the corresponding string in $I \cup E$.

Fig. 5 compares the normal suffix tree and the word suffix tree for string $w = \mathbf{ab}\#\mathbf{ab}\#\mathbf{a}\#$, where $\Sigma = \{\mathbf{a}, \mathbf{b}\}$ and $D = \Sigma^* \cdot \#$.

4.2 Word Suffix Tree Construction Algorithm

Note that $|I| + |E| = O(k)$, which means that the size of $WSTree_D(w)$ is also $O(k)$. Since $WSTree_D(w)$ is path compressed, the edges of $WSTree_D(w)$ are labeled by substrings of w rather than single characters. By implementing these substring labels with pointers to w , $WSTree_D(w)$ can be finally implemented in $O(k)$ space. The time cost for word suffix tree construction is $\Omega(n)$ due to the need of scanning the whole string w . Thus, the final goal is to construct $WSTree_D(w)$ in $O(n)$ time with $O(k)$ space.

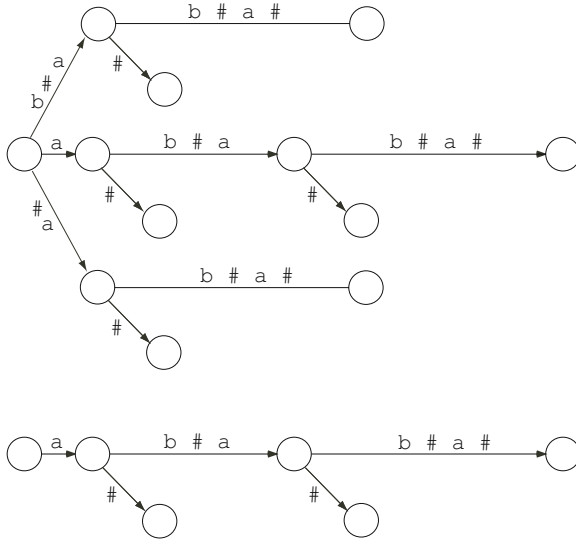


Fig. 5. The normal suffix tree of $w = ab\#ab\#a\#$ on the upper, and the word suffix tree of w w.r.t. $D = \{a, b\}^* \cdot \#$ on the lower.

Suffix Link. The suffix links of $WSTree_D(w)$ are a key to achieve the above goal. Recall that any node s of $WSTrie_D(w)$ is regarded as a unique ordered pair $\langle x, y \rangle$, such that $x \in D^*$ and y is a proper prefix of some string in D . We apply the same notion to the nodes of $WSTree_D(w)$. Also, we use the auxiliary DFA M_D that accepts D in the same way.

Definition 5 (Suffix links of word suffix tree). Let $D = \Sigma^* \cdot \#$ and M_D be the smallest DFA that accepts D . For each node $s = \langle x, y \rangle$ of $WSTree_D(w)$,

1. if $s \in I$ and $x \in D^+$, the suffix link from s goes to node $\langle x', y \rangle$ such that $x' \in D^*$ and $x = hx'$ for some $h \in D$;
2. if $s \in I$ and $x = \varepsilon$, the suffix link from s goes to the initial state of M_D ;
3. otherwise (if $s \in E$), the suffix link from s is undefined.

The suffix links of those in Group 3 in the above definition remain undefined, as they are never used in our construction algorithm to be shown later. See Fig. 6

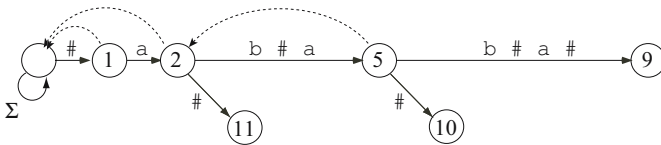


Fig. 6. The word suffix tree with auxiliary DFA M_D and suffix links (broken arrows), where $w = ab\#ab\#a\#$ and $D = \{a, b\}^* \cdot \#$. Note that the suffix links of nodes 9, 10, and 11, which are those in Group 3 of Definition 5, are missing, as they are never used in the construction algorithm.

```

Input:     $w = w[1..n] \in D^+$  and auxiliary DFA  $M_D$ .
Output:  Word suffix tree of  $w[1..n]$  w.r.t.  $D$ .
{
   $root =$  the final state of  $M_D$ ;  $slink(root) =$  the initial state of  $M_D$ ;
   $(s, k) = (root, 1)$ ;
  for  $(i = 1; i \leq n; i++)$  {
     $olldr = \text{nil}$ ;
    while  $(CheckEndPoint(s, (k, i - 1), w[i]) == \text{false})$  {
      if  $(k \leq i - 1)$   $r = SplitEdge(s, (k, i - 1))$ ;
      else  $r = s$ ;
       $t = CreateNewNode()$ ;
      create a new edge  $r \xrightarrow{(i, \infty)} t$ ;
      if  $(olldr \neq \text{nil})$   $slink(olldr) = r$ ;
       $olldr = r$ ;
       $(s, k) = Canonize(slink(s), (k, i - 1))$ ;
    }
    if  $(olldr \neq \text{nil})$   $slink(olldr) = s$ ;
     $(s, k) = Canonize(s, (k, i))$ ;
  }
}

boolean  $CheckEndPoint(s, (k, p), c)$  {
  if  $(k \leq p)$  { /*  $(s, (k, p))$  is implicit. */
    let  $s \xrightarrow{(k', p')} s'$  be the  $w[k]$ -edge from  $s$ ;
    return  $(c == w[k' + p - k + 1])$ ;
  } else return (there is a  $c$ -edge from  $s$ );
}

(node, integer)-pair  $Canonize(s, (k, p))$  {
  if  $(k > p)$  return  $(s, k)$ ; /*  $(s, (k, p))$  is explicit. */
  find the  $w[k]$ -edge  $s \xrightarrow{(k', p')} s'$  from  $s$ ;
  while  $(p' - k' \leq p - k)$  {
     $k += p' - k' + 1$ ;  $s = s'$ ;
    if  $(k \leq p)$  find the  $w[k]$ -edge  $s \xrightarrow{(k', p')} s'$  from  $s$ ;
  }
  return  $(s, k)$ ;
}

node  $SplitEdge(s, (k, p))$  {
  let  $s \xrightarrow{(k', p')} s'$  be the  $w[k]$ -edge from  $s$ ;
   $r = CreateNewNode()$ ;
  replace this edge by edges  $s \xrightarrow{(k', k' + p - k)} r$  and  $r \xrightarrow{(k' + p - k + 1, p')} s'$ ;
  return  $r$ ;
}

```

Fig. 7. Word suffix tree construction algorithm

for the word suffix tree with the auxiliary DFA M_D and suffix links, using the running example.

Algorithm. A pseudo-code of our algorithm to build word suffix trees is summarized in Fig. 7. It simulates construction of word suffix tries in $O(n)$ time and with $O(k)$ space. Fig. 8 shows a snapshot of on-line construction of $WSTree_D(w)$ with the running example.

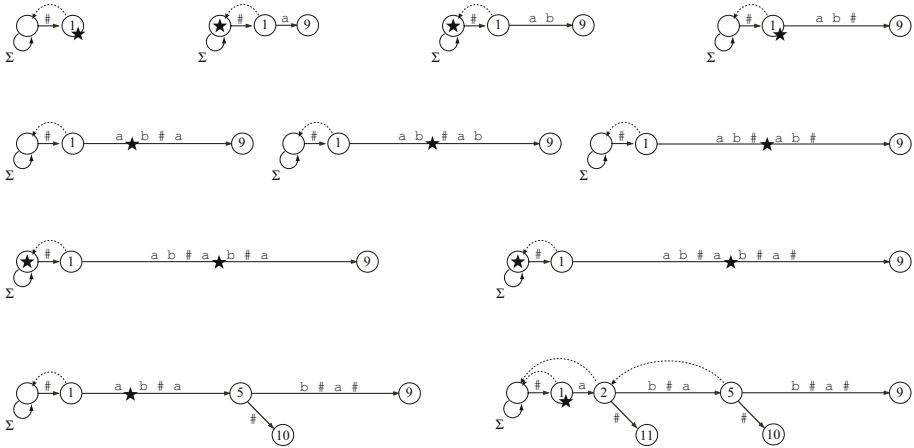


Fig. 8. A snapshot of on-line construction of $WSTree_D(w)$ with $w = ab\#ab\#a\#$ and $D = \{a, b\}^* \cdot \#$. The update with the last $\#$ is shown in three steps. The star mark denotes the location represented by $(s, (k, i - 1))$ in the algorithm of Fig. 7, from which a new edge is possibly created.

The main result of this paper follows:

Theorem 1. *The algorithm of Fig. 7 builds word suffix trees in linear time (on a fixed alphabet).*

Proof. Since the algorithm is a time and space economical simulation of the word suffix trie construction algorithm of Fig. 3, the correctness follows from Lemma 1.

We now prove the linearity of the algorithm. Consider the location, referred to as $(s, (k, i - 1))$, which represents the substring $w[k - \ell..i - 1]$ where ℓ is the length of the string represented by the node s . One iteration of the **while** loop in the main routine alters s into $slink(s)$ and therefore decreases the length of the substring by at least one. We note that *Canonize* never alters the substring represented by $(s, (k, p))$ although it might update s and k . On the other hand, the length of the substring is increased by at most one at each iteration of the **for** loop in the main routine. Thus, the total number of iterations of the **while** loop in the main routine is linearly proportional to the input string length. We have only to estimate the total cost of all executions of *Canonize*. We note that the

value of variable k changes only by an execution of *Canonize*, and monotonically increases. The cost of one execution of *Canonize* is proportional to the number of iterations of the **while** loop in it plus one, which is linear with respect to the number of times the variable k is increased during the iterations. The total cost of all executions of *Canonize* is therefore proportional to the number of times k is increased in the execution of the algorithm. Since the length of the string $w[k..i - 1]$ is increased by at most one at each iteration of the **for** loop in the main routine, the number of times k is increased is linear with respect to the input string length. \square

5 Conclusions and Further Discussions

We have presented a new on-line algorithm for constructing word suffix trees. The algorithm is very simple and runs in linear time *even in the worst cases*, whereas the one proposed by Anderson et al. runs in linear time *on the average*.

The simplicity of our algorithm is due to the use of DFA M_D accepting a dictionary D . The idea comes from the synchronization technique introduced in [17] in which similar DFA are embedded onto the Aho-Corasick pattern matching machines [1] so that they process multi-byte character texts in a byte-by-byte manner without extra work for avoiding false matches.

Lastly, our algorithm can be seen as a practical solution to efficient construction of general sparse suffix trees. Let $w \in \Sigma^*$ and Pos be a set of positions of suffixes we want to store in the sparse suffix tree. Let $|w| = n$ and $|Pos| = k$. For any position i in Pos , we insert the special character $\#$ at position $i - 1$ of the original string w . Note that the length of the modified string w' is at most twice as that of the original string w , and therefore the word suffix tree for w' can be constructed with $O(k)$ space in $O(n)$ time. To search for pattern $p \in \Sigma^*$ of length m , we skip any $\#$ in the word suffix tree of w' . This way the matching can be done correctly, and in $O(m)$ time.

References

1. A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.
2. A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.
3. A. Apostolico. The myriad virtues of subword trees. *Combinatorial Algorithms on Words*, F12:85–96, 1985.
4. R. Baeza-Yates and G. H. Gonnet. Efficient text searching of regular expressions. In *Proc. 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, volume 372 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1989.
5. H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano. Efficiently finding regulatory elements using correlation with gene expression. *Journal of Bioinformatics and Computational Biology*, 2(2):273–288, 2004.

6. R. Clifford and M. Sergot. Distributed and paged suffix trees for large genetic databases. In *Proc. 14th Ann. Symp. on Combinatorial Pattern Matching (CPM'03)*, volume 2676 of *Lecture Notes in Computer Science*, pages 70–82. Springer-Verlag, 2003.
7. B. Dorohonceanu and C. G. Nevill-Manning. Accelerating protein classification using suffix trees. In *Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB'00)*, pages 128–133. AAAI Press, 2000.
8. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
9. S. Inenaga, H. Bannai, H. Hyrö, A. Shinohara, M. Takeda, K. Nakai, and S. Miyano. Finding optimal pairs of cooperative and competing patterns with bounded distance. In *Proc. 7th International Conference on Discovery Science (DS'04)*, volume 3245 of *Lecture Notes in Artificial Intelligence*, pages 32–46. Springer-Verlag, 2004.
10. S. Inenaga, T. Funamoto, M. Takeda, and A. Shinohara. Linear-time off-line text compression by longest-first substitution. In *Proc. 10th International Symp. on String Processing and Information Retrieval (SPIRE'03)*, volume 2857 of *Lecture Notes in Computer Science*, pages 137–152. Springer-Verlag, 2003.
11. S. Inenaga, T. Kivioja, and V. Mäkinen. Finding missing patterns. In *Proc. 4th Workshop on Algorithms in Bioinformatics (WABI'04)*, volume 3240 of *Lecture Notes in Bioinformatics*, pages 463–474. Springer-Verlag, 2004.
12. J. Kärkkänen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd International Computing and Combinatorics Conference (COCOON'96)*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer-Verlag, 1996.
13. N. J. Larsson. Extended application of suffix trees to data compression. In *Proc. Data Compression Conference '96 (DCC'96)*, pages 190–199. IEEE Computer Society, 1996.
14. L. Marsan and M.-F. Sagot. Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification. In *Proc. 4th Annual International Conference on Computational Molecular Biology (RECOMB'00)*, pages 210–219. ACM, 2000.
15. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.
16. J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1–3):87–101, 2003.
17. M. Takeda, S. Miyamoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Processing text files as is: Pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In *Proc. 9th International Symp. on String Processing and Information Retrieval (SPIRE'02)*, volume 2476 of *Lecture Notes in Computer Science*, pages 170–186. Springer-Verlag, 2002.
18. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
19. P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.