On-Line Construction of Suffix Trees[1]

E. Ukkonen[2]

Abstract. An on-line algorithm is presented for constructing the suffix tree for a given string in time linear in the length of the string. The new algorithm has the desirable property of processing the string symbol by symbol from left to right. It always has the suffix tree for the scanned part of the string ready. The method is developed as a linear-time version of a very simple algorithm for (quadratic size) suffix *tries*. Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long. Another variation of this method is shown to give, in a natural way, the well-known algorithms for constructing suffix automata (DAWGs).

Key Words. Linear-time algorithm, Suffix tree, Suffix trie, Suffix automaton, DAWG.

1. Introduction. A *suffix tree* is a trie-like data structure representing all suffixes of a string. Such trees have a central role in many algorithms on strings, see, e.g., [3], [7], and [2]. It is quite commonly felt, however, that the linear-time suffix tree algorithms presented in the literature are rather difficult to grasp.

The main purpose of this paper is an attempt to develop an understandable suffix tree construction based on a natural idea that seems to complete our picture of suffix trees in an essential way. The new algorithm has the important property of being on-line. It processes the string symbol by symbol from left to right, and always has the suffix tree for the scanned part of the string ready. The algorithm is based on the simple observation that the suffixes of a string $T^i = t_1 \cdots t_i$ can be obtained from the suffixes of string $T^{i-1} = t_1 \cdots t_{i-1}$ by catenating symbol t_i at the end of each suffix of T^{i-1} and by adding the empty suffix. The suffixes of the whole string $T = T^n = t_1 t_2 \cdots t_n$ can be obtained by first expanding the suffixes of T^0 into the suffixes of T^1 and so on, until the suffixes of T are obtained from the suffixes of T^{n-1}.

This is in contrast with the method by Weiner [13] that proceeds right to left and adds the suffixes to the tree in increasing order of their length, starting from the shortest suffix, and with the method by McCreight [9] that adds the suffixes to the tree in decreasing order of their length. It should be noted, however, that despite the clear difference in the intuitive view on the problem, our algorithm and McCreight's algorithm are in their final form functionally rather closely related.

[2] Department of Computer Science, University of Helsinki, P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland. ukkonen@cs.Helsinki.FI.

Our algorithm is best understood as a linear-time version of another algorithm from [12] for (quadratic-size) suffix *tries*. The latter very elementary algorithm, which resembles the position tree algorithm in [8], is given in Section 2. Unfortunately, it does not run in linear time—it takes time proportional to the size of the suffix trie which can be quadratic. However, a rather transparent modification, which we describe in Section 4, gives our on-line, linear-time method for suffix trees. This also offers a natural perspective which makes the linear-time suffix tree construction understandable.

We also point out in Section 5 that the suffix trie augmented with the suffix links gives an elementary characterization of the suffix automata (also known as directed acyclic word graphs or DAWGs). This immediately leads to an algorithm for constructing such automata. Fortunately, the resulting method is essentially the same as already given in [4]–[6]. Again it is felt that our new perspective is very natural and helps in understanding the suffix automata constructions.

2. Constructing Suffix Tries.

Let $T = t_1 t_2 \cdots t_n$ be a string over an alphabet Σ. Each string x such that $T = uxv$ for some (possibly empty) strings u and v is a *substring* of T, and each string $T_i = t_i \cdots t_n$ where $1 \le i \le n + 1$ is a *suffix* of T; in particular, $T_{n+1} = \varepsilon$ is the *empty* suffix. The set of all suffixes of T is denoted $\sigma(T)$. The *suffix trie* of T is a trie representing $\sigma(T)$.

More formally, we denote the suffix trie of T as $STrie(T) = (Q \cup \{\bot\}, root, F, g, f)$ and define such a trie as an augmented DFA (deterministic finite-state automaton) which has a tree-shaped transition graph representing the trie for $\sigma(T)$ and which is augmented with the so-called suffix function f and auxiliary state $\bot$. The set Q of the states of $STrie(T)$ can be put in a one-to-one correspondence with the substrings of T. We denote by $\bar{x}$ the state that corresponds to a substring x.

The initial state *root* corresponds to the empty string ε, and the set F of the final states corresponds to $\sigma(T)$. The transition function g as defined as $g(\bar{x}, a) = \bar{y}$ for all $\bar{x}, \bar{y}$ in Q such that $y = xa$, where $a \in \Sigma$.

The *suffix function* f is defined for each state $\bar{x} \in Q$ as follows. Let $\bar{x} \ne root$. Then $x = ay$ for some $a \in \Sigma$, and we set $f(\bar{x}) = \bar{y}$. Moreover, $f(root) = \bot$.

Auxiliary state $\bot$ allows us to write the algorithms in what follows such that an explicit distinction between the empty and the nonempty suffixes (or, between *root* and the other states) can be avoided. State $\bot$ is connected to the trie by $g(\bot, a) = root$ for every $a \in \Sigma$. We leave $f(\bot)$ undefined. (Note that the transitions from $\bot$ to *root* are defined consistently with the other transitions: State $\bot$ corresponds to the inverse a^{-1} of all symbols $a \in \Sigma$. Because $a^{-1}a = \varepsilon$, we can set $g(\bot, a) = root$ as *root* corresponds to ε.)

Following [9] we call $f(r)$ the *suffix link* of state r. The suffix links are utilized during the construction of a suffix tree; they also have many uses in the applications (e.g., [11] and [12]).

Automaton $STrie(T)$ is identical to the Aho–Corasick string matching automaton [1] for the key-word set $\{T_i | 1 \le i \le n + 1\}$ (the suffix links are called *failure transitions* in [1]).
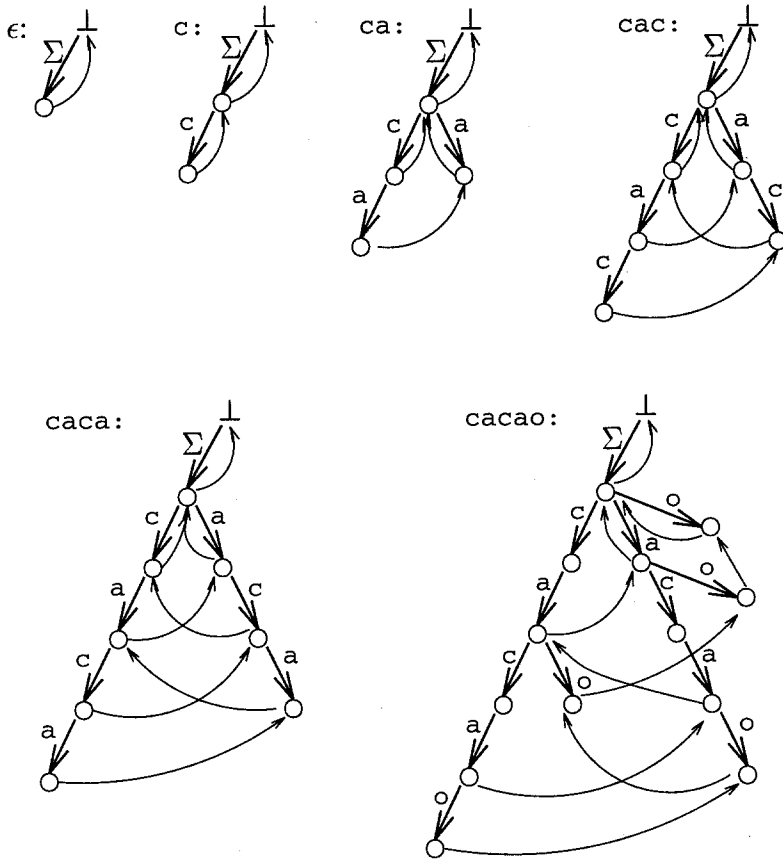
Fig. 1. Construction of *STrie*(cacao): state transitions are shown by bold arrows, failure transitions by thin arrows. *Note*: Only the last two layers of suffix links are shown explicitly.

It is easy to construct *STrie*(T) on-line, in a left-to-right scan over T, as follows. Let T^i denote the *prefix* $t_1 \cdots t_i$ of T for $0 \le i \le n$. As intermediate results the construction gives *STrie*(T^i) for $i = 0, 1, \ldots, n$. Figure 1 shows the different phases of constructing *STrie*(T) for $T = $ cacao.

The key observation explaining how *STrie*(T^i) is obtained from *STrie*(T^{i-1}) is that the suffixes of T^i can be obtained by catenating t_i to the end of each suffix of T^{i-1} and by adding an empty suffix. That is,

$$\sigma(T^i) = \sigma(T^{i-1})t_i \cup \{\varepsilon\}.$$

By definition, *STrie*(T^{i-1}) accepts $\sigma(T^{i-1})$. To make it accept $\sigma(T^i)$, we must examine the final state set F_{i-1} of *STrie*(T^{i-1}). If $r \in F_{i-1}$ does not already have a t_i-transition, then such a transition from r to a new state (which becomes a new leaf of the trie) is added. The states to which there is an old or new t_i-transition from some state in F_{i-1} constitute together with *root* the final states F_i of *STrie*(T^i).

The states $r \in F_{i-1}$ that get new transitions can be found using the suffix links as follows. The definition of the suffix function implies that $r \in F_{i-1}$ if and only if $r = f^j(\overline{t_1 \cdots t_{i-1}})$ for some $0 \le j \le i-1$. Therefore, all states in F_{i-1} are on the path of suffix links that starts from the deepest state $\overline{t_1 \cdots t_{i-1}}$ of $STrie(T^{i-1})$ and ends at $\perp$. We call this important path the *boundary path* of $STrie(T^{i-1})$.

The boundary path is traversed. If a state $\bar{z}$ on the boundary path does not have a transition on t_i yet, a new state $\overline{zt_i}$ and a new transition $g(\bar{z}, t_i) = \overline{zt_i}$ are added. This gives updated g. To get updated f, the new states $\overline{zt_i}$ are linked together with new suffix links that form a path starting from state $\overline{t_1 \cdots t_i}$. Obviously, this is the boundary path of $STrie(T^i)$.

The traversal over F_{i-1} along the boundary path can be stopped immediately the *first* state $\bar{z}$ is found such that state $\overline{zt_i}$ (and hence also transition $g(\bar{z}, t_i) = \overline{zt_i}$) already exists. To see this, let $\overline{zt_i}$ already be a state. Then $STrie(T^{i-1})$ has to contain state $\overline{z't_i}$ and transition $g(\overline{z'}, t_i) = \overline{z't_i}$ for all $\overline{z'} = f^j(\bar{z}), j \ge 1$. In other words, if zt_i is a substring of T^{i-1}, then every suffix of zt_i is a substring of T^{i-1}. Note that $\bar{z}$ always exists because $\perp$ is the last state on the boundary path and $\perp$ has a transition for every possible t_i.

When the traversal is stopped in this way, the procedure creates a new state for every suffix link examined during the traversal. This implies that the whole procedure takes time proportional to the size of the resulting automaton.

Summarized, the procedure for building $STrie(T^i)$ from $STrie(T^{i-1})$ is as follows [12]. Here *top* denotes the state $\overline{t_1 \cdots t_{i-1}}$.

Algorithm 1
> $r \leftarrow top$;
> **while** $g(r, t_i)$ is undefined **do**
> > create new state r' and new transition $g(r, t_i) = r'$;
> > **if** $r \ne top$ **then** create new suffix link $f(oldr') = r'$;
> > $oldr' \leftarrow r'$;
> > $r \leftarrow f(r)$;
>
> create new suffix link $f(oldr') = g(r, t_i)$;
> $top \leftarrow g(top, t_i)$.

Starting from $STrie(\varepsilon)$, which consists only of *root* and $\perp$ and the links between them, and repeating Algorithm 1 for $t_i = t_1, t_2, \ldots, t_n$, we obviously get $STrie(T)$. The algorithm is optimal in the sense that it takes time proportional to the size of its end result $STrie(T)$. This in turn is proportional to $|Q|$, that is, to the number of different substrings of T. Unfortunately, this can be quadratic in $|T|$, as is the case, for example, if $T = a^n b^n$.

THEOREM 1. *Suffix trie $STrie(T)$ can be constructed in time proportional to the size of $STrie(T)$ which, in the worst case, is $O(|T|^2)$.*

3. Suffix Trees. Suffix tree $STree(T)$ of T is a data structure that represents $STrie(T)$ in space linear in the length $|T|$ of T. This is achieved by representing only a subset $Q' \cup \{\perp\}$ of the states of $STrie(T)$. We call the states in $Q' \cup \{\perp\}$

the *explicit states*. Set Q' consists of all *branching states* (states from which there are at least two transitions) and all *leaves* (states from which there are no transitions) of $STrie(T)$. By definition, *root* is included in the branching states. The other states of $STrie(T)$ (the states other than *root* and $\perp$ from which there is exactly one transition) are called *implicit states* as states of $STree(T)$; they are not explicitly present in $STree(T)$.

The string w spelled out by the transition path in $STrie(T)$ between two explicit states s and r is represented in $STree(T)$ as generalized transition $g'(s, w) = r$. To save space the string w is actually represented as a pair (k, p) of pointers (the *left pointer* k and the *right pointer* p) to T such that $t_k \cdots t_p = w$. In this way the generalized transition gets the form $g'(s, (k, p)) = r$.

Such pointers exist because there must be a suffix T_i such that the transition path for T_i in $STrie(T)$ goes through s and r. We could select the smallest such i, and let k and p point to the substring of this T_i that is spelled out by the transition path from s to r. A transition $g'(s, (k, p)) = r$ is called an *a-transition* if $t_k = a$. Each s can have at most one *a*-transition for each $a \in \Sigma$.

Transitions $g(\perp, a) = root$ are represented in a similar fashion as follows. Let $\Sigma = \{a_1, a_2, \ldots, a_m\}$. Then $g(\perp, a_j) = root$ is represented as $g(\perp, (-j, -j)) = root$ for $j = 1, \ldots, m$.

Hence suffix tree $STree(T)$ has two components: the tree itself and the string T. It is of linear size in $|T|$ because Q' has at most $|T|$ leaves (there is at most one leaf for each nonempty suffix) and therefore Q' has to contain at most $|T| - 1$ branching states (when $|T| > 1$). There can be at most $2|T| - 2$ transitions between the states in Q', each taking a constant space because of the use of pointers instead of an explicit string. (Here we have assumed the standard RAM model in which a pointer takes constant space.)

We again augment the structure with the suffix function f', now defined only for all branching states $\bar{x} \neq root$ as $f'(\bar{x}) = \bar{y}$ where y is a branching state such that $x = ay$ for some $a \in \Sigma$, and $f'(root) = \perp$. Such an f' is well defined: if $\bar{x}$ is a branching state, then $f'(\bar{x})$ is also a branching state. These suffix links are explicitly represented. It is sometimes helpful to speak about *implicit* suffix links, i.e., imaginary suffix links between the implicit states.

The suffix tree of T is denoted as $STree(T) = (Q' \cup \{\perp\}, root, g', f')$.

We refer to an explicit or implicit state r of a suffix tree by a *reference pair* (s, w) where s is some explicit state that is an ancestor of r and w is the string spelled out by the transitions from s to r in the corresponding suffix trie. A reference pair is *canonical* if s is the closest ancestor of r (and, hence, w is the shortest possible). For an explicit r the canonical reference pair obviously is (r, ε). Again, we represent string w as a pair (k, p) of pointers such that $t_k \cdots t_p = w$. In this way a reference pair (s, w) gets the form $(s, (k, p))$. Pair (s, ε) is represented as $(s, (p + 1, p))$.

It is technically convenient to omit the final states in the definition of a suffix tree. When explicit final states are needed in some application, they are obtained gratuitously by adding to T an end marking symbol that does not occur elsewhere in T. The leaves of the suffix tree for such a T are in one-to-one correspondence with the suffixes of T and constitute the set of the final states.

Another possibility is to traverse the suffix link path from leaf $\bar{T}$ to *root* and make all states on the path explicit; these states are the final states of $STree(T)$. In many applications of $STree(T)$, the start location of each suffix is stored with the corresponding state. Such an augmented tree can be used as an index for finding any substring of T.

4. On-Line Construction of Suffix Trees.

The algorithm for constructing $STree(T)$ is patterned after Algorithm 1. What has to be done is for the most part immediately clear. Figure 2 shows the phases of constructing $STree(\text{cacao})$; for simplicity, the strings associated with each transition are shown explicitly in the figure. However, to get a linear-time algorithm some details need more careful examination.

We first make more precise what Algorithm 1 does. Let $s_1 = \overline{t_1 \cdots t_{i-1}}$, $s_2, s_3, \ldots, s_i = root$, $s_{i+1} = \perp$ be the states of $STrie(T^{i-1})$ on the boundary path. Let j be the smallest index such that s_j is not a leaf, and let j' be the smallest index such that $s_{j'}$ has a t_i-transition. As s_1 is a leaf and $\perp$ is a nonleaf that has a t_i-transition, both j and j' are well defined and $j \leq j'$. Now the following lemma should be obvious.

LEMMA 1.   *Algorithm 1 adds to $STrie(T^{i-1})$ a t_i-transition for each of the states s_h, $1 \leq h < j'$, such that, for $1 \leq h < j$, the new transition expands an old branch of the trie that ends at leaf s_h, and, for $j \leq h < j'$, the new transition initiates a new branch from s_h. Algorithm 1 does not create any other transitions.*

We call state s_j the *active point* and $s_{j'}$ the *endpoint* of $STrie(T^{i-1})$. These states are present, explicitly or implicitly, in $STree(T^{i-1})$, too. For example, the active points of the last three trees in Figure 2 are $(root, \text{c})$, $(root, \text{ca})$, $(root, \varepsilon)$.
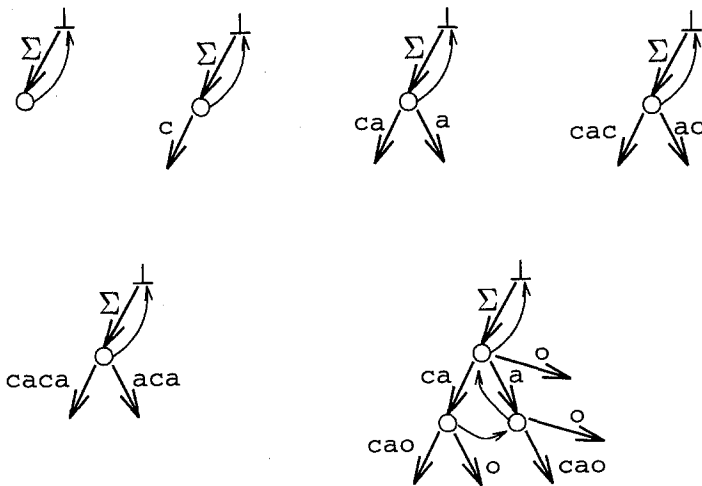

Fig. 2. Construction of $STree(\text{cacao})$.

Lemma 1 says that Algorithm 1 inserts two different groups of t_i-transitions into $STrie(T^{i-1})$:

(i) The states on the boundary path before the active point s_j get a transition. These states are leaves, hence each such transition has to expand an existing branch of the trie.

(ii) The states from the active point s_j to the endpoint $s_{j'}$, the endpoint is excluded, get a new transition. These states are not leaves, hence each new transition has to initiate a new branch.

We next interpret this in terms of suffix tree $STree(T^{i-1})$. The first group of transitions that expand an existing branch could be implemented by updating the right pointer of each transition that represents the branch. Let $g'(s, (k, i-1)) = r$ be such a transition. The right pointer has to point to the last position $i-1$ of T^{i-1}. This is because r is a leaf and therefore a path leading to r has to spell out a suffix of T^{i-1} that does not occur elsewhere in T^{i-1}. Then the updated transition must be $g'(s, (k, i)) = r$. This only makes the string spelled out by the transition longer but does not change the states s and r. Making all such updates would take too much time. Therefore, we use the following trick.

Any transition of $STree(T^{i-1})$ leading to a leaf is called an *open transition*. Such a transition is of the form $g'(s, (k, i-1)) = r$ where, as stated above, the right pointer has to point to the last position $i-1$ of T^{i-1}. Therefore it is not necessary to represent the actual value of the right pointer. Instead, open transitions are represented as $g'(s, (k, \infty)) = r$ where ∞ indicates that this transition is "open to grow." In fact, $g'(s, (k, \infty)) = r$ represents a branch of *any* length between state s and the imaginary state r that is "in infinity." An explicit updating of the right pointer when t_i is inserted into this branch is not needed. Symbols ∞ can be replaced by $n = |T|$ after completing $STree(T)$. In this way the first group of transitions is implemented without any explicit changes to $STree(T^{i-1})$.

We still have to describe how to add the second group of transitions to $STree(T^{i-1})$. These create entirely new branches that start from states s_h, $j \le h < j'$. Finding such states s_h needs some care as they need not be explicit states at the moment. They are found along the boundary path of $STree(T^{i-1})$ using reference pairs and suffix links.

Let $h = j$ and let (s, w) be the canonical reference pair for s_h, i.e., for the active point. As s_h is on the boundary path of $STrie(T^{i-1})$, w has to be a suffix of T^{i-1}. Hence $(s, w) = (s, (k, i-1))$ for some $k \le i$.

We want to create a new branch starting from the state represented by $(s, (k, i-1))$. However, first we test whether or not $(s, (k, i-1))$ already refers to the endpoint $s_{j'}$. If it does, we are done. Otherwise a new branch has to be created. To this end the state s_h referred to by $(s, (k, i-1))$ has to be explicit. If it is not, an explicit state, denoted s_h, is created by splitting the transition that contains the corresponding implicit state. Then a t_i-transition from s_h is created. It has to be an open transition $g'(s_h, (i, \infty)) = s'_h$ where s'_h is a new leaf. Moreover, the suffix link $f'(s_h)$ is added if s_h was created by splitting a transition.

Next the construction proceeds to s_{h+1}. As the reference pair for s_h was $(s, (k, i-1))$, the canonical reference pair for s_{h+1} is $canonize(f'(s), (k, i-1))$ where

canonize makes the reference pair canonical by updating the state and the left pointer (note that the right pointer $i - 1$ remains unchanged in canonization). The above operations are then repeated for s_{h+1}, and so on until the endpoint $s_{j'}$ is found.

In this way we obtain the procedure *update*, given below, that transforms $STree(T^{i-1})$ into $STree(T^i)$ by inserting the t_i-transitions in the second group. The procedure uses procedure *canonize* mentioned above, and procedure *test-and-split* that tests whether or not a given reference pair refers to the endpoint. If it does not, then the procedure creates and returns an explicit state for the reference pair provided that the pair does not already represent an explicit state. Procedure *update* returns a reference pair for the endpoint $s_{j'}$ (actually only the state and the left pointer of the pair, as the second pointer remains $i - 1$ for all states on the boundary path).

procedure *update*$(s, (k, i))$:
   $(s, (k, i - 1))$ is the canonical reference pair for the active point;
1. *oldr* ← *root*; (*end-point*, r) ← *test-and-split*$(s, (k, i - 1), t_i)$;
2. **while not**(*end-point*) **do**
3.   create new transition $g'(r, (i, \infty)) = r'$ where r' is a new state;
4.   **if** *oldr* ≠ *root* **then** create new suffix link $f'(oldr) = r$;
5.   *oldr* ← r;
6. (s, k) ← *canonize*$(f'(s), (k, i - 1))$;
7.   (*end-point*, r) ← *test-and-split*$(s, (k, i - 1), t_i)$;
8. **if** *oldr* ≠ *root* **then** create new suffix link $f'(oldr) = s$;
9. **return** (s, k).

Procedure *test-and-split* tests whether or not a state with canonical reference pair $(s, (k, p))$ is the endpoint, that is, a state that in $STrie(T^{i-1})$ would have a t_i-transition. Symbol t_i is given as input parameter t. The test result is returned as the first output parameter. If $(s, (k, p))$ is not the endpoint, then state $(s, (k, p))$ is made explicit (if not already so) by splitting a transition. The explicit state is returned as the second output parameter.

procedure *test-and-split*$(s, (k, p), t)$:
1. **if** $k \leq p$ **then**
2.   let $g'(s, (k', p')) = s'$ be the t_k-transition from s;
3.   **if** $t = t_{k' + p - k + 1}$ **then return**(**true**, s)
4.   **else**
5.   replace the t_k-transition above by transitions
              $g'(s, (k', k' + p - k)) = r$ and $g'(r, (k' + p - k + 1, p')) = s'$
              where r is a new state;
6.   **return**(**false**, r)
7. **else**
8.   **if** there is no t-transition from s **then return**(**false**, s)
9.   **else return**(**true**, s).

This procedure benefits from the fact that $(s, (k, p))$ is canonical: the answer to the endpoint test can be found in constant time by considering only one transition from s.

Procedure *canonize* is as follows. Given a reference pair $(s, (k, p))$ for some state r, it finds and returns state s' and left link k' such that $(s', (k', p))$ is the canonical reference pair for r. State s' is the closest explicit ancestor of r (or r itself if r is explicit). Therefore the string that leads from s' to r must be a suffix of the string $t_k \cdots t_p$ that leads from s to r. Hence the right link p does not change but the left link k can become $k', k' \geq k$.

> **procedure** *canonize*$(s, (k, p))$;
> 1. **if** $p < k$ **then return** (s, k)
> 2. **else**
> 3.   find the t_k-transition $g'(s, (k', p')) = s'$ from s;
> 4.   **while** $p' - k' \leq p - k$ **do**
> 5. $k \leftarrow k + p' - k' + 1$;
> 6. $s \leftarrow s'$;
> 7.   **if** $k \leq p$ **then** find the t_k-transition $g'(s, (k', p')) = s'$ from s;
> 8.   **return** (s, k).

To be able to continue the construction for the next text symbol t_{i+1}, the active point of $STree(T^i)$ has to be found. To this end, note first that s_j is the active point of $STree(T^{i-1})$ if and only if $s_j = \overline{t_j \cdots t_{i-1}}$ where $t_j \cdots t_{i-1}$ is the longest suffix of T^{i-1} that occurs at least twice in T^{i-1}. Second, note that $s_{j'}$ is the endpoint of $STree(T^{i-1})$ if and only if $s_{j'} = \overline{t_{j'} \cdots t_{i-1}}$ where $t_{j'} \cdots t_{i-1}$ is the longest suffix of T^{i-1} such that $t_{j'} \cdots t_{i-1}t_i$ is a substring of T^{i-1}. However, this means that if $s_{j'}$ is the endpoint of $STree(T^{i-1})$, then $t_{j'} \cdots t_{i-1}t_i$ is the longest suffix of T^i that occurs at least twice in T^i, that is, then state $g(s_{j'}, t_i)$ is the active point of $STree(T^i)$.

We have shown the following result.

LEMMA 2. *Let $(s, (k, i-1))$ be a reference pair of the endpoint $s_{j'}$ of $STree(T^{i-1})$. Then $(s, (k, i))$ is a reference pair of the active point of $STree(T^i)$.*

The overall algorithm for constructing $STree(T)$ is finally as follows. String T is processed symbol by symbol in one left-to-right scan. Writing $\Sigma = \{t_{-1}, \ldots, t_{-m}\}$ makes it possible to present the transitions from $\perp$ in the same way as the other transitions.

> **Algorithm 2.** Construction of $STree(T)$ for string $T = t_1 t_2 \cdots \#$ in alphabet $\Sigma = \{t_{-1}, \ldots, t_{-m}\}$; $\#$ is the end marker not appearing elsewhere in T.
> 1. create states *root* and $\perp$;
> 2. **for** $j \leftarrow 1, \ldots, m$ **do** create transition $g'(\perp, (-j, -j)) = root$;
> 3. create suffix link $f'(root) = \perp$;
> 4. $s \leftarrow root$; $k \leftarrow 1$; $i \leftarrow 0$;
> 5. **while** $t_{i+1} \neq \#$ **do**
> 6. $i \leftarrow i + 1$;
> 7. $(s, k) \leftarrow update(s, (k, i))$;
> 8. $(s, k) \leftarrow canonize(s, (k, i))$.

Steps 7–8 are based on Lemma 2: after step 7 pair $(s, (k, i - 1))$ refers to the endpoint of $STree(T^{i-1})$, and, hence, $(s, (k, i))$ refers to the active point of $STree(T^i)$.

THEOREM 2. *Algorithm 2 constructs the suffix tree $STree(T)$ for a string $T = t_1 \cdots t_n$ on-line in time $O(n)$.*

PROOF. The algorithm constructs $STree(T)$ through intermediate trees $STree(T^0)$, $STree(T^1), \ldots, STree(T^n) = STree(T)$. It is on-line because to construct $STree(T^i)$ it only needs access to the first i symbols of T.

For the running-time analysis we divide the time requirement into two components, both turn out to be $O(n)$. The first component consists of the total time for procedure *canonize*. The second component consists of the rest: the time for repeatedly traversing the suffix link path from the present active point to the endpoint and creating the new branches by *update* and then finding the next active point by taking a transition from the endpoint (step 8 of Algorithm 2). We call the states (reference pairs) on these paths the *visited states*.

The second component takes time proportional to the total number of the visited states, because the operations (create an explicit state and a new branch, follow an explicit or implicit suffix link, test for the endpoint) at each such state can be implemented in constant time as *canonize* is excluded. (To be precise, this also requires that $|\Sigma|$ is bounded independently of n.) Let r_i be the active point of $Stree(T^i)$ for $0 \leq i \leq n$. The visited states between r_{i-1} and r_i are on a path that consists of some suffix links and one t_i-transition. Taking a suffix link decreases the *depth* (the length of the string spelled out on the transition path from *root*) of the current state by one, and taking a t_i-transition increases it by one. The number of the visited states (including r_{i-1}, excluding r_i) on the path is therefore $depth(r_{i-1}) - depth(r_i) + 2$, and their total number is $\sum_{i=1}^{n} (depth(r_{i-1}) - depth(r_i) + 2) = depth(r_0) - depth(r_n) + 2n \leq 2n$. This implies that the second time component is $O(n)$.

The time spent by each execution of *canonize* has an upper bound of the form $a + bq$ where a and b are constants and q is the number of executions of the body of the loop in steps 5–7 of *canonize*. The total time spent by *canonize* has therefore a bound that is proportional to the sum of the number of the calls of *canonize* and the total number of the executions of the body of the loop in all calls. There are $O(n)$ calls as there is one call for each visited state (either in step 6 of *update* or directly in step 8 of Algorithm 2). Each execution of the body deletes a nonempty string from the left end of string $w = t_k \cdots t_p$ represented by the pointers in reference pair $(s, (k, p))$. String w can grow during the whole process only in step 8 of Algorithm 2 which catenates t_i for $i = 1, \ldots, n$ to the right end of w. Hence a nonempty deletion is possible at most n times. The total time for the body of the loop is therefore $O(n)$, and altogether *canonize* or our first component needs time $O(n)$.   □

REMARK 1 (due to J. Kärkkäinen). In its final form our algorithm is a rather close relative of McCreight's method [9]. The principal technical difference seems to be that each execution of the body of the main loop of our Algorithm 2 consumes one text symbol t_i, whereas each execution of the body of the main loop of McCreight's algorithm traverses one suffix link and consumes zero or more text symbols.

REMARK 2. It is not hard to generalize Algorithm 2 for the following dynamic version of the suffix tree problem (cf., the *adaptive dictionary matching problem* of [2]): Maintain a generalized linear-size suffix tree representing all suffixes of strings T_i in set $\{T_1, \ldots, T_k\}$ under operations that insert or delete a string T_i. The resulting algorithm will make such updates in time $O(|T_i|)$.

5. Constructing Suffix Automata.

The *suffix automaton* $SA(T)$ of a string $T = t_1 \cdots t_n$ is the minimal DFA that accepts all the suffixes of T.

As our $STrie(T)$ is a DFA for the suffixes of T, $SA(T)$ could be obtained by minimizing $STrie(T)$ in the standard way. Minimization works by combining the equivalent states, i.e., states from which $STrie(T)$ accepts the same set of strings. Using the suffix links we obtain a natural characterization of the equivalent states as follows.

A state s of $STrie(T)$ is called *essential* if there is at least two different suffix links pointing to s or $s = \overline{t_1 \cdots t_k}$ for some k.

THEOREM 3. *Let s and r be two states of $STrie(T)$. The set of strings accepted from s is equal to the set of strings accepted from r if and only if the suffix link path that starts from s contains r (the path from r contains s) and the subpath from s or r (from r to s) does not contain any other essential states than possibly s (r).*

PROOF. The theorem is implied by the following observations.

The set of strings accepted from some state of $STrie(T)$ is a subset of the suffixes of T and therefore each accepted string is of different length.

A string of length i is accepted from a state s of $STrie(T)$ if and only if the suffix link path that starts from state $\overline{t_1 \cdots t_{n-i}}$ contains s.

The suffix links form a tree that is directed to its root *root*.   □

This suggests a method for constructing $SA(T)$ with a modified Algorithm 1. The new feature is that the construction should create a new state only if the state is essential. An unessential state s is merged with the first essential state that is before s on the suffix link path through s. This is correct as, by Theorem 3, the states are equivalent.

As there are $O(|T|)$ essential states, the resulting algorithm can be made to work in linear time. The algorithm turns out to be similar to the algorithms in [4]–[6]. We therefore omit the details.

Acknowledgments. J. Kärkkäinen pointed out some inaccuracies in the earlier version [10] of this work. The author is also indebted to E. Sutinen, D. Wood, and, in particular, S. Kurtz and G. A. Stephen for several useful comments.

References

[1]   A. Aho and M. Corasick, Efficient string matching: an aid to bibliographic search, *Comm. ACM*, **18** (1975), 333–340.

[2]   A. Amir and M. Farach, Adaptive dictionary matching, *Proc. 32nd IEEE Ann. Symp. on Foundations of Computer Science*, 1991, pp. 760–766.

[3]   A. Apostolico, The myriad virtues of subword trees, in *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), Springer-Verlag, New York, 1985, pp. 85–95.

[4]   A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.*, **40** (1985), 31–55.

[5]   M. Crochemore, Transducers and repetitions, *Theoret. Comput. Sci.*, **45** (1986), 63–86.

[6]   M. Crochemore, String matching with constraints, in *Mathematical Foundations of Computer Science 1988* (M. P. Chytil, L. Janiga and V. Koubek, eds.), Lecture Notes in Computer Science, vol. 324, Springer-Verlag, Berlin, 1988, pp. 44–58.

[7]   Z. Galil and R. Giancarlo, Data structures and algorithms for approximate string matching, *J. Complexity*, **4** (1988), 33–72.

[8]   M. Kempf, R. Bayer, and U. Güntzer, Time optimal left to right construction of position trees, *Acta Inform.*, **24** (1987), 461–474.

[9]   E. McCreight, A space-economical suffix tree construction algorithm, *J. Assoc. Comput. Mach.*, **23** (1976), 262–272.

[10] E. Ukkonen, Constructing suffix trees on-line in linear time, in *Algorithms, Software, Architecture. Information Processing 92*, vol. I (J. van Leeuwen, ed.), Elsevier, Amsterdam, 1992, pp. 484–492.

[11] E. Ukkonen, Approximate string-matching over suffix trees, in *Combinatorial Pattern Matching, CPM '93* (A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds.), Lecture Notes in Computer Science, vol. 684, Springer-Verlag, Berlin 1993, pp. 228–242.

[12] E. Ukkonen and D. Wood, Approximate string matching with suffix automata, *Algorithmica*, **10** (1993), 353–364.

[13] P. Weiner, Linear pattern matching algorithms, *Proc. IEEE 14th Ann. Symp. on Switching and Automata Theory*, 1973, pp. 1–11.