𝕋ℂ𝕊-𝕋ℝ-𝔸-**11-51**

𝕋ℂ𝕊 Technical Report

Sparse and Truncated Suffix Trees on Variable-Length Codes

by

Takashi Uemura and Hiroki Arimura

Division of Computer Science

Report Series A

March 28, 2011

Hokkaido University
Graduate School of
Information Science and Technology

Email: arim@ist.hokudai.ac.jp

Phone: +81-011-706-7680
Fax: +81-011-706-7680

Sparse and Truncated Suffix Trees on Variable-Length Codes

Takashi Uemura[1] and Hiroki Arimura[1]

Graduate School of Infomration Science and Technology, Hokkaido University
Kita 14, Nishi 9, Kita-ku, Sapporo, 060-0814 Japan
{tue,arim}@ist.hokudai.ac.jp

Abstract. The sparse suffix trees (SST), introduced by (Kärkkäinen and Ukkonen, *COCOON 1996*), is the suffix tree for a subset of all suffixes of an input text T of length n. In this paper, we study a special case that an input string is a sequence of codewords drawn from a regular prefix code $\Delta \subseteq \Sigma^+$ recognized by a finite automaton, and index points locate on the code boundaries. In this case, we present an online algorithm that constructs the sparse suffix tree for an input string t on any variable-length regular prefix code, called the code suffix tree (CST), in $O(n+m)$ time and $O(k)$ additional space for a fixed base alphabet Σ, where m is the size of an automaton for Δ. Furthermore, we present a modified algorithm for k-truncated version of code suffix trees that runs in the same time and space complexities. Hence, these results generalize the previous results (Inenaga and Takeda, *CPM 2006*) for word suffix trees and (Na, Apostolico, Iliopoulos, and Park, *Theor. Comp. Sci.*, 304, 2003) for truncated suffix trees on arbitrary variable-length regular prefix codes.

1 Introduction

Backgrounds The *sparse suffix trees* (SST), introduced by Kärkkäinen and Ukkonen [11] in 1996, is the suffix tree (ST) [6, 13, 17] for storing a subset consisting of k suffixes in an input text T of length n on a base alphabet Σ, where $k \leq n$. In its most general form, the set $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ of *index points* which is given as an arbitrary subset of all n text positions. We denote by $\mathsf{SST}^I(T)$ the sparse suffix tree for T with respect to the set I of index points. [11] showed that a sparse suffix tree on a k-evenly indexed string in $O(n)$ worst-case time and $O(k)$ space. Although a sparse suffix tree for a string with an arbitrary index set is well-defined in any sense, interestingly enough, it is still open since its introduction whether a sparse suffix tree for arbitrary set I of k index positions can be constructed in $O(n)$ time and $O(k)$ additional space.

For the problem, recently, a collection of word-based suffix indexes have been introduced [3, 5, 8–10]. To formalize this notion, we introduce the set $\Delta_I(T) \subseteq \Sigma^+$ of words, called the *induced code*, obtained by partitioning an input text T by the index positions in I. Then, a suffix index is called *word-based* if the set $\Delta_I(T)$ is restricted to a set of words in $\Sigma^+ W$, where W is a finite set of symbols, called word delimiters, such that $W \cap \Sigma = \emptyset$. In 1999, Andersson, Larsson, and Swanson [3] introduced a word-suffix tree as a $\mathsf{SST}^{I}(T)$ on a word alphabet, and presented a construction algorithm in $O(n)$ average time and $O(k)$ space. In 2006, Inenaga and Takeda [8] presented the first construction algorithm that runs in $O(n)$ worst-case time and $O(k)$ space by modifying Ukkonen's linear time online construction algorithm for full suffix trees [17]. Their work is most closely related to this work. Ferragina and Fischer [5] introduced word-suffix arrays and presented a construction algorithm with $O(n)$ worst-case time and $O(k)$ space.

Our contribution. In this paper, we study the sparse suffix tree construction in more general setting than that of the word-based suffix trees [3, 8]. In particular, we consider the sparse suffix tree for a string on an arbitrary regular prefix code $\Delta \subseteq \Sigma^{+}$ which is recognized by a finite deterministic automaton. The sparse suffix tree of this type is called the *code suffix tree* (CST), and a natural generalization of word suffix trees [3, 8]. As a main result of this paper, we show that the code suffix tree for an input string T of length n on a prefix code Δ can be constructed in $O(n + m)$ worst-case time and $O(k)$ space for a fixed base alphabet Σ, where k is the number of words in T and m is the size of the automaton for Δ (Theorem 2).

Key techniques. To show this, we propose a modified version of Ukkonen's online suffix tree construction algorithm [17] augmented with a DFA, called a *code automaton*, for recognizing Δ, which is similar to the construction in [8]. However, the proofs for correctness and time complexity are not straightforward due to the complex behavior of the algorithm when it traverses inside of a code automaton. To overcome this difficulty, we introduce an extended domain of strings augmented with the erasing element $\perp$, which is the inverse of any codeword in Δ and acting from left. Using $\perp$, we give a general definition of suffix links, and show that most properties of full suffix trees [17], including the existence lemma for suffix links, still remains valid when Δ is a prefix code. Hence, Theorem 2 above gives a partial answer to a natural question: what is the largest class of codes for which the approach of Ukkonen's linear-time construction algorithm [17] is sufficient for constructing sparse suffix trees on a code?

An extension. For every $\ell \geq 1$, an *ℓ-truncated suffix tree* (ℓ-TST) for T is a variation of suffix trees that stores all factors of T with length ℓ. Na *et al.* [14] introduced ℓ-TST and presented an online construction algorithm in $O(n)$ time and space. Based on the algorithm for CST, we present a modified version of the algorithm for *ℓ-truncated code suffix trees* (ℓ-TCST) that constructs ℓ-TCST for a text on Δ in $O(n)$ worst-case time and $O(k)$ space, where k is the number of words in T (Theorem 3). Finally, we ran experiments on real datasets to evaluate the usefulness of the proposed methods. For example, CSTs are 3 to 5 times smaller than STs on English and UTF-8 texts.

Organization of this paper. In Sec. 2, we give basic definitions. In Sec. 3, we present our linear-time construction algorithm $\mathsf{ConstructCST}$ for code suffix trees. In Sec. 4, we extend the algorithm for ℓ-truncated code suffix trees. In Sec. 5, we show experimental results, and in Sec. 6, we conclude this paper.

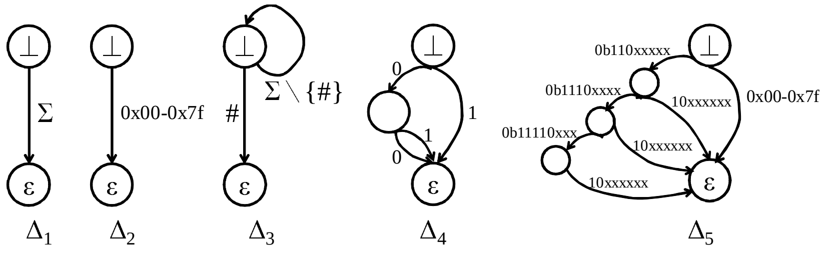

Fig. 1. The code automata for prefix codes $\Delta_1, \Delta_2, \Delta_3, \Delta_4$, and Δ_5 in Example 1, where $\bot$ and ε are the initial and the final states, respectively, and labels with wildcard $x \in \{0, 1\}$, e.g., $0x00 - 0x7f$, represent sets of the corresponding multiple edges.

2 Preliminaries

Basic definitions. We introduce basic definitions on suffix trees according to [4, 6, 11, 14, 17]. We assume that the reader has basic knowledge of the linear time construction algorithm by Ukkonen [17]. Let Σ be an alphabet of *base letters*. We denote by ε the *empty string*. Let Σ^* and Σ^+ denote the *sets of all possibly empty finite strings* and *non-empty finite strings* on Σ. For a string T, if $T = xyz$ for some $x, y, z \in \Sigma^*$, then we call x, y, and z a *prefix*, a *factor* (*substring*), and a *suffix* of T, respectively. Let $T = a_1 \cdots a_n \in \Sigma^*$ be a string on Σ of length $|T|_\Sigma = |T| = n$, where $T[i] = a_i \in \Sigma$ is the *i-th letter* for every $i = 1, \ldots, n$. For any $1 \leq i \leq j \leq n$, we denote by $T[i..j] = a_i \cdots a_j$ the *factor from i to j* of T. If $i > j$ then we define $T[i..j] = \varepsilon$. For a set $\mathcal{S} \subseteq \Sigma^*$ of strings, the *sets of the prefixes* and *proper prefixes* of all strings in $\mathcal{S}$ are denoted by $\mathsf{Pre}(\mathcal{S})$ and by $\mathsf{PropPre}(\mathcal{S})$, respectively. $|\mathcal{S}|$ and $||\mathcal{S}||$ denote the *cardinality* and the *total size* of $\mathcal{S}$. For strings x, y, we denote by $\mathsf{lcp}(x, y)$ the *longest common prefix* of x and y.

Prefix Codes. A *code* is a set $\Delta \subseteq \Sigma^+$, where each $w \in \Delta$ is a non-empty string, called a *codeword* (or *word*, for short) of Δ. A *preword* is any prefix $u \in \mathsf{Pre}(\Delta)$ and a *proper preword* is any proper prefix $u \in \mathsf{PropPre}(\Delta)$ of a word in Δ. A code Δ is either infinite or finite. Δ is a *prefix code* if it is *prefix-free*, that is, any codeword is not a prefix of some other codeword of Δ.

Example 1. Let Σ be a letter alphabet and $\mathbb{B} = \{1, 0\}$ be a binary alphabet. A trivial prefix code $\Delta_1 = \Sigma$ and the *ASCII code* $\Delta_2 = [0x00 - 0x7f] \subseteq \mathbb{B}^8$ are prefix codes of fixed-length. For a word delimitor $\sharp \not\in \Sigma$, a *word alphabet* $\Delta_3 = \Sigma^+\sharp$ is an example of prefix-codes of variable-length ([3, 5, 8–10]). A *Huffman code* $\Delta_4 = \{00, 01, 1\}$ for the set of symbols $\{A, B, C\}$ with probabilities $p(A) = 1/4$, $p(B) = 1/4$, $p(C) = 1/2$ ([4]), and the code $\Delta_5 = (10\mathbb{B}^1) \cup (110\mathbb{B}^2) \cup (1110\mathbb{B}^3)$, called the *three-byte fragment of UTF-8* ([7]), are also examples of prefix-codes of variable-length. As seen later, all of these prefix codes are regular (Fig. 1).

Codeword strings. A *word string* on Δ (or *Δ-string*) is a string $T \in \Delta^*$. Then, an *input string* or a *pre-string* of letter length $n \geq 0$ is any prefix $T =$

$T[1]\cdots T[n] \in \mathsf{Pre}(\Delta^*)$ of a word string on Δ, where $T[i] \in \Sigma$ for $i = 1, \dots, n$. Since Δ is a prefix code and thus $\mathsf{Pre}(\Delta^*) = \Delta^* \mathsf{PropPre}(\Delta)$, we have the unique Δ-factoring $T = w_1 \cdots w_k w_{k+1} \in \mathsf{Pre}(\Sigma^*)$ of T, where $w_1, \dots, w_k \in \Delta$ and $w_{k+1} \in \mathsf{PropPre}(\Delta)$. Then, the proper preword w_{k+1} is called the *tail* of T. Clearly, T is a complete word string if and only if the tail w_{k+1} is emtpy. We define the *word length* of T by $|T|_\Delta = k$, and the *letter length* by $|T|_\Sigma = |T| = n$. where $n = \sum_{i=1}^{k+1} |w_i|$.

In this paper, construction of a CST is done in online manner through the stage $i = 0, \dots, n$ as in [15, 17]. We define the current input $T^i = T[1] \cdots T[i] = w_1 \cdots w_k w_{k+1}$, where $k \geq 0$, $w_{j'} \in \Delta$, and $w_{k+1} \in \mathsf{PropPre}(\Delta)$. Let $j = 1, \dots, k+1$ be any index. The j-th *Δ-suffix* of T is defined as the suffix of T starting at the j-th word boundary, i.e., $\mathsf{suf}_j^\Delta(T) = w_j \cdots w_k w_{k+1}$. For $\ell \geq 1$, the j-th *(Δ, ℓ)-factor* of T is defined as the factor $\mathsf{fac}_j^{\Delta,\ell}(T) = w_j \cdots w_h$ of T, where $h = \min\{j + \ell - 1, k+1\}$. We denote by $\mathsf{Suf}^\Delta(T)$ and $\mathsf{Fac}^\Delta(T, \ell)$ the *sets of all Δ-suffixes* and *all (Δ, ℓ)-factors* of T, respectively.

Code suffix trees. A *code suffix tree* (CST, for short) for an input string $T \in \mathsf{Pre}(\Delta^*)$ w.r.t. a prefix code Δ, denoted by $\mathsf{CST}^\Delta(T)$, is a compacted trie [6] that represents all Δ-suffixes of T. Formally, the CST for T is a rooted tree $S = \mathsf{CST}^\Delta(T) = (V, \mathit{child}, \mathit{root}, \mathit{Label}(\cdot), \mathit{SL}(\cdot))$ that satisfies the following properties. V is a finite set of *tree nodes* (or *nodes*). Each directed edge $e = (u, v)$ is labeled with a factor of T, $\mathit{Label}(v) \in \Sigma^+$, stored in v. Every internal node except the root is *branching*, i.e., it has at least two children. For every base letter $a \in \Sigma$, each internal node v has at most one directed edge from v to the *a-child*, $u = \mathit{child}(v, a)$, whose label starts with a. $\mathit{SL}(\cdot)$ is a *suffix link* function, which will be defined later in Section 3. For each v, we denote by $L(v)$ the string *represented by* v, that is, the string obtained by concatenating all labels on the path from the root to v. All the suffixes are represented by the leaves of $\mathsf{CST}^\Delta(T)$ when T ends with the unique marker $T[n]$ that does not appear elsewhere. All the factors in T that start at word boundaries are represented as the prefixes of all leaves, that is, the elements of the set $\mathsf{str}(\text{tree}) = \mathsf{Pre}(\mathsf{Suf}^\Delta(\text{T}))$. We give the naming function $[\cdot]$ below. For any $\alpha \in \mathsf{str}(\text{tree})$, we define the *locus* of α, denoted by $[\alpha]$, to be the unique tree node $v \in Q$ such that $L(v) = \alpha$. This mapping $[\cdot]$ is one-one.

Clearly, $\mathsf{CST}^\Delta(T)$ has at most k leaves and $k-1$ internal nodes [11]. To save the space, we represent $\mathit{Label}(v)$ by a pair $\langle j, i \rangle \in \mathbb{N}^2$ of the starting and ending positions of the label in T such that $T[j..i] = \mathit{Label}(v)$, while ε is represented by $\langle i+1, i \rangle$ for some i. Assuming this, $\mathsf{CST}^\Delta(T)$ occupies only $O(k)$ space. For any factor α of T that starts at a word boundary, we represent its location in $\mathsf{CST}^\Delta(T)$ by a triple $p = \langle v, j, i \rangle \in V \times \mathbb{N}^2$, called a *pointer* (or a *reference*) to α, such that $\alpha = L(v) \cdot T[j..i]$ if it exists. A pointer $\langle v, j, i \rangle$ is *canonical* if $T[j..i]$ is shortest. The *locus* of a factor α is the canonical pointer for α, and denoted by $\mathit{loc}(\alpha)$. We often call p a *virtual node* if $j \leq i$, i.e., $T[j..i] \neq \varepsilon$, and *real node* if $j > i$, i.e., $T[j..i] = \varepsilon$.

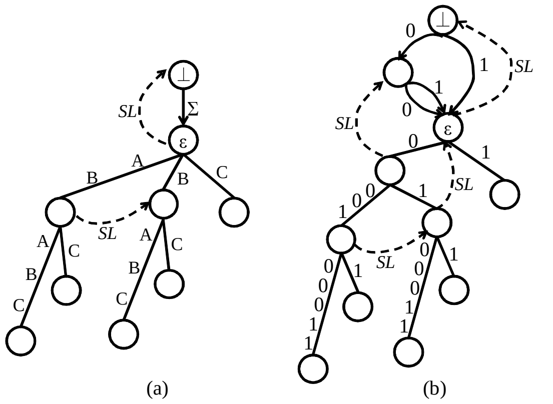

Fig. 2. Examples of (a) an ordinary suffix tree for string $S = ABABC$ on alphabet $\{A, B, C\}$, and (b) a code suffix tree for coded string $T = 000100011$ on prefix code $\varDelta = \{A/00, B/01, C/1\}$.

3 A Linear-time Online Algorithm for Code Suffix Trees

In this section, we show our algorithm $\mathsf{ConstructCST}$ for constructing a code suffix tree $\mathsf{CST}^{\varDelta}(T)$ on a prefix code $\varDelta$, which is based on Ukkonen's online construction algorithm for suffix trees [17]. The only difference is that it is augmented with a *code automaton* and *code suffix links* explained below. Let us fix an input string $T = w_1 \cdots w_k w_{k+1} \in \mathsf{Pre}(\varDelta^*)$ on a prefix code $\varDelta \subseteq \varSigma^+$ with letter length n and word length k.

3.1 Code automata and code suffix links

Code automata. In our problem setting, a prefix code $\varDelta \subseteq \varSigma^+$ on $\varSigma$ is *regular* if $\varDelta$ is recognized by a finite automaton. A *code automaton* for a prefix code $\varDelta \subseteq \varSigma^+$ is a possibly cyclic deterministic finite automaton (DFA) $A = (\varSigma, Q, \mathit{child}, \hat{\bot}, \mathit{root})$ on a base alphabet $\varSigma$, where Q is a finite set of *code nodes* (or *nodes*). $\hat{\bot}$ and *root* are the unique initial and final states, called the *source* and *sink*, respectively. The function $\mathit{child} : Q \times \varSigma \to Q$ is a *transition function* such that for every $u, v \in Q$ and $a \in \varSigma$, $\mathit{child}(u, a) = v$ if and only if there exists an a-edge from u to v labeled with a. We extend *child* to a mapping $\mathit{child}^* : Q \times \varSigma^* \to Q$ in a standard way [4]. If it is clear from context, we refer to the code automaton A for $\varDelta$ as $\mathsf{DFA}(\varDelta)$. In the treatment of this paper, $\mathsf{DFA}(\varDelta)$ need not be minimal in general. As a related work, Takeda, Miyamoto *et al.* [15] used code automata to extend Aho-Corasick pattern matching machines on a prefix code. For example, we show in Fig. 1 the code automata for codes $\varDelta_1$, $\varDelta_2$, $\varDelta_3$, $\varDelta_4$, and $\varDelta_5$ in Example. 1.

We give the naming function $[\cdot]$ below. We introduce a special element $\bot \notin \varSigma$ as the *inverse* of any word of $\varDelta$, i.e., $\bot w = \varepsilon$ for every $w \in \varDelta$. We define $\bot w\alpha = \varepsilon\alpha = \alpha$ if $w \in \varDelta$ and $\alpha \in \varSigma^*$. For a proper preword $\alpha \in \mathsf{PropPre}(\varDelta)$, $\bot\alpha$ is a special element different from α. For any $\alpha \in \mathsf{str}(\mathrm{code}) = \mathsf{Pre}(\Delta)$, we define the *locus* of α, denoted by $[\bot\alpha]$, to be the unique code node $v \in Q$ such that

$child^*(\hat{\bot}, \alpha) = v$. Note that the mapping $[\cdot]$ is many-one and naturally induces an equivalence relation $\equiv$ on the set $\mathsf{str}(\text{code})$. Since $\mathsf{DFA}(\Delta)$ has the unique final state *root* $= [\varepsilon]$, we see that $[\bot\alpha] = [\bot\beta] = [\varepsilon]$ holds for any codewords $\alpha, \beta \in \Delta$. We also note that $[\bot\alpha] \neq [\alpha]$ for any string α since the former is the node reachable from $\hat{\bot}$ in $\mathsf{DFA}(\Delta)$ and the latter is the node reachable from *root* in $\mathsf{CST}^{\Delta}(T)$. As a note, the above notations are just for analysis of our algorithm, and do not affect the behavior and complexity of the algorithm.

By the above encoding, we can represent $\mathsf{DFA}(\Delta)$ as follows: The node set is $Q = \{\, [\bot\alpha] \mid \alpha \in \mathsf{str}(\text{code}) \,\}$. The transition function is given by $child([\alpha], a) = [\alpha a]$ for every $\alpha \in \mathsf{str}(\text{code})$ and $a \in \Sigma$. The source and the sink are $[\bot]$ and $[\varepsilon]$, respectively. If $[\alpha]$ is either a tree node or a code node and $a \in \Sigma$ is any letter, then we define $[\alpha]\cdot a = [\alpha \cdot a]$. We define the set $\mathsf{dom}(\text{tree}) = \{[\alpha] \mid \alpha \in \mathsf{str}(\text{tree})\}$ of all tree nodes, and $\mathsf{dom}(\text{code}) = \{\, [\bot\alpha] \mid \alpha \in (\mathsf{str}(\text{code}) - \Delta) \,\} = \mathrm{Q} - \{[\varepsilon]\}$ of code nodes. Furthermore, we define $\mathsf{dom}(\text{pre}) = \{\, [\alpha] \mid \alpha \in \mathsf{PropPre}(\Delta) \,\} \subseteq \mathsf{dom}(\text{tree})$, the domain of all *prenodes*. By definition, $[\varepsilon] \in \mathsf{dom}(\text{tree})$ but $[\varepsilon] \notin \mathsf{dom}(\text{code})$. In what follows, we often use α and $[\alpha]$ interchangeably if no confusion arises.

Code suffix links. Next, we introduce the suffix links for $\mathsf{CST}^{\Delta}(T)$ as follows. Similarly to Ukkonen's algorithm, each internal node $v = [\alpha]$ in the CST has the suffix link of v, denoted by $SL^{\Delta}(v)$, which is a pointer from v to the internal node u such that $SL^{\Delta}([\alpha]) = [\bot \cdot \alpha]$, where $\alpha \in \mathsf{str}(\text{tree})$. Equivalently, if $v = [w\beta]$ for some $w \in \Delta, \beta \in \mathsf{str}(\text{tree}) \cup \mathsf{str}(\text{code})$ then $SL^{\Delta}([w\beta]) = [\beta]$. Any code node $v \in \mathsf{dom}(\text{code})$ does not have a suffix link. The next lemma is crucial to the correctness of our algorithm.

Lemma 1 (existence lemma for code suffix links). *Let $\Delta \subseteq \Sigma^+$ be a prefix-free code and T be any pre-string on Δ. Then, (i) any tree node v in $\mathsf{CST}^{\Delta}(T)$ has the suffix link $SL^{\Delta}(v)$ pointing to a branching internal node u in either $\mathsf{CST}^{\Delta}(T)$ or $\mathsf{DFA}(\Delta)$. Furthermore, (ii) v is a preword node if and only if $SL^{\Delta}(v)$ is a code node in $\mathsf{DFA}(\Delta)$, and (iii) v is not a preword node if and only if $SL^{\Delta}(v)$ is a tree node in $\mathsf{CST}^{\Delta}(T)$.*

Proof. There are two cases on the domain of v. (1) If $v = [\alpha] \in \mathsf{dom}(\text{pre})$ is a prenode, then $SL^{\Delta}(v) = [\bot\alpha]$ belongs to $\mathsf{dom}(\text{code})$ by the definition of $\mathsf{DFA}(\Delta)$. (2) Suppose that $= [w\alpha] \in \mathsf{dom}(\text{tree})\backslash\mathsf{dom}(\text{pre})$ for some $w \in \Delta, \alpha \in \Sigma^*$. It is shown in [11] that a SST has a branching node v if and only if $L(v) = \mathsf{lcp}(\mathsf{suf}^{\Delta}_i(T), \mathsf{suf}^{\Delta}_j(T))$ for some indexes i and j. Since Δ is prefix-free and $v = [w\alpha]$, we know that both of $\mathsf{suf}^{\Delta}_i(T)$ and $\mathsf{suf}^{\Delta}_j(T)$ start with w, and thus, we have $\alpha = \mathsf{lcp}(\mathsf{suf}^{\Delta}_{i+1}(T), \mathsf{suf}^{\Delta}_{j+1}(T))$. From the above claim, the lemma follows. □

3.2 Main algorithm

In Fig. 3, we show the algorithm $\mathsf{ConstructCST}$, and in Fig. 4, the subprocedures Extend and $\mathsf{Terminate}$. The only difference between our algorithm and Ukkonen's algorithm is lines 3 and 4 of $\mathsf{ConstructCST}$ that attaches $\mathsf{DFA}(\Delta)$ to the CST. For an input string $T = T[1] \cdots T[n] = w_1 \cdots w_k w_{k+1} \in \mathsf{Pre}(\Delta^*)$ on $\Delta \subseteq \Sigma^+$,

```
**Algorithm** ConstructCST:
**input:** A preword string $T = w_1 \cdots w_k \in \mathsf{Pre}(\Delta^*)$ on a prefix code $\Delta \subseteq \Sigma$;
**output:** The sparse suffix tree $\mathsf{CST}^\Delta(T)$ for $t$ w.r.t. $\Delta$. ;
{ **global variables:** $\Theta, \Theta'$: word counters //for $\ell$-TCST }
Create an empty tree $\mathsf{CST}^\Delta$ with the root node $root = [\varepsilon]$;
Build $\mathsf{DFA}(\Delta)$ for $\Delta$ with the source $\hat{\bot} = [\bot]$ and the sink $root = [\varepsilon]$;
$SL^\Delta(root) = \hat{\bot}$;
$\phi \leftarrow \langle root, 1, 0\rangle$; $\psi \leftarrow \langle root, 1, 0\rangle$;
{ $\mathsf{Reset}(\Theta)$; $\mathsf{Reset}(\Theta')$ //for $\ell$-TCST }
**for** $i = 1, \dots, n$ **do** //Stage $i$
    $\phi \leftarrow \mathsf{Extend}(\phi, i)$;
    {$\psi \leftarrow \mathsf{Terminate}(\psi, i)$; //for $\ell$-TCST}
**end for**
**return** $\mathsf{CST}^\Delta$;
```

Fig. 3. A construction algorithm for a code suffix tree $\mathsf{CST}^\Delta(T)$ for a text T on a prefix code $\Delta \subseteq \Sigma^+$.

the algorithm constructs the CST for $T_i = T[1..i]$ in an online manner for every stage $i = 1, \dots, n$. At stage 0, the CST consists only of the root node $root = [\varepsilon]$ and $\mathsf{DFA}(\Delta)$. Let $T^i = T[1..i]$ be the current input text and $\mathsf{CST}^\Delta(T^i)$ be the code suffix tree for T^i obtained at the end of stage i. At each step i, the algorithm extends α to the new suffixes αa_i by appending the current base letter $a_i = T[i]$ for all Δ-suffixes in $\mathsf{CST}^\Delta(T^i)$.

This extension process is based on the following idea. Let $S^\Delta(i) = Suf^\Delta(T^i)$ be the set of all Δ-suffixes in T^i. For every stage $i = 0, \dots, n$, we define the set $Bd^\Delta(i) \subseteq \mathsf{dom}(\text{tree}) \cup \mathsf{dom}(\text{code})$, the *border*, by the following recurrence:

- $Bd^\Delta(0) = \{\bot, \varepsilon\}$,
- $Bd^\Delta(i) = (Bd^\Delta(i-1) \cdot a_i) \ \cup \ \{\bot\}$ if $\varepsilon \in (Bd^\Delta(i-1) \cdot a_i)$,
- $Bd^\Delta(i) = Bd^\Delta(i-1) \cdot a_i$ otherwise,

where $S \cdot a = \{\, \alpha a \,|\, \alpha \in S \,\}$ for any set $S \subseteq \Sigma^*$ and $a \in \Sigma$. Then, we have the following lemma. Recall that $\mathsf{dom}(\text{tree})$ is the domain of tree nodes in $\mathsf{CST}^\Delta(T^i)$.

Lemma 2. *For every $i = 0, \dots, n$, $S^\Delta(i) = Bd^\Delta(i) \cap \mathsf{dom}(\text{tree})$.*

Proof. From a similar argument to [15, 17], the following recurrence holds:

(i) $S^\Delta(0) = \{\varepsilon\}$,
(ii) $S^\Delta(i) = (S^\Delta(i-1) \cdot a_i) \ \cup \ \{\varepsilon\}$ if $T^i = T[1..i]$ is a complete word string (*),
(iii) $S^\Delta(i) = S^\Delta(i-1) \cdot a_i$ otherwise,

By induction on $i \geq 0$, we then can show that the condition (*) holds iff $w \in S^\Delta(i-1) \cdot a_i$ for some $w \in \Delta$ iff $\varepsilon \in Bd^\Delta(i-1) \cdot a_i$ holds. Furthermore, $\bot\alpha \in Bd^\Delta(i)$ iff $\alpha \in S^\Delta(i)$ for any $\alpha \in \mathsf{Pre}(\Delta)$. Thus, the result follows. □

```
procedure Extend($\phi = \langle s, j, i-1\rangle, i$):
  $last \leftarrow NULL$; //oldp
  while ($child(\phi, T[i])$ is not defined) do
    if $j \le i$ then begin
      $\phi \leftarrow$ Split($\phi$);
      if $last \neq NULL$ then begin
        $SL^{\Delta}(\phi) \leftarrow last$;
        $last \leftarrow \phi$; end
    end
    create a leaf $q$ with $label(q) \leftarrow \langle i, \infty\rangle$;
    $child(\phi, T[i]) \leftarrow q$;
    $\phi \leftarrow$ Canonize($\langle SL^{\Delta}(s), j, i-1\rangle$);
    {Decrement($\Theta$) //for $\ell$-TST}
  end while
  $\phi \leftarrow$ Canonize($\langle s, j, i\rangle$);
  {ChildTrans($\Theta, T[i]$); //for $\ell$-TST}
  {if toClose($\Theta$) then SuffixTrans($\Theta, \phi$);
    //for $\ell$-TST}
  return $\phi$; {End of Extend}

procedure Canonize($\phi = \langle s, j, i\rangle$):
  while $j \le i$ do begin
    $u \leftarrow child(s, T[j])$;
    $\langle q, p\rangle \leftarrow label(u)$;
    if $p - q > i - j$ then
      break;
    $j \leftarrow j + (p - q + 1)$;
    $s \leftarrow u$;
  end
  return $\langle s, j, i\rangle$; {End of Canonize}
```

Fig. 4. The subprocedures Extend and Canonize for the code suffix tree construction.

We call each suffix α in $Bd^{\Delta}(i)$ the *extension point* at stage i. Next, we consider how we can efficiently find the extension points and extend them. The detection of ε is also crucial to the synchronization of word boundaries. We use a pointer ϕ to keep track to extension points in $Bd^{\Delta}(i)$ from longer to shorter. Let act_i^{Δ} be the *active point* at stage i as the pointer ϕ such that $L(\phi)$ is the longest suffix of T^{i-1} that occurs at least twice in T^i. For $i = 1, \ldots, n$, the maintenance of $Bd^{\Delta}(i)$ proceeds in the following way for all extension points $\phi = [\alpha a_i]$ of three types 1–3. Let $Bd^{\Delta}(0) = \{\perp, \varepsilon\}$.

- *type 1*: If α occurs only at the end of T^{i-1}, then, α is represented by a leaf, and thus so is αa_i. As in Ukkonen [17], by representing α as an open leaf $\langle j, \infty\rangle$, ∞ is interpreted as the current index i, the extension point α is automatically extended without any management. This correctly extends all extension points α of type 1.
- *type 2*: If α occurs at least twice in T^{i-1}, but αa_i does not occur in T^i, then by induction on i, we can show that act_i^{Δ} is the first node of type 2 satisfying this condition. Then, we create the new node for αa_i extending α by appending a to the tail of α, while the parent α is materialized by procedure Split if α is virtual. Repeat this process until we reach the first node ϕ of type 3. This correctly extends all extension points α of type 2.
- *type 3*: If α occurs in T^{i-1} at least twice in T^{i-1}, and αa_i also occurs in T^i, then we can show as in Ukkonen [17] that all extension points on the suffix links from α to some node in $\mathsf{dom}(\text{code}) \cup \{[\varepsilon]\}$, which is the end of the border, are already contained in T^i. Therefore, these extension points are correctly extended in $\mathsf{CST}^{\Delta}(T^i)$ without any explicit extension.

The procedure Extend in Fig. 4 implements the above incremental computation of $Bd^{\Delta}(i)$. From the discussion above, we have the next lemma.

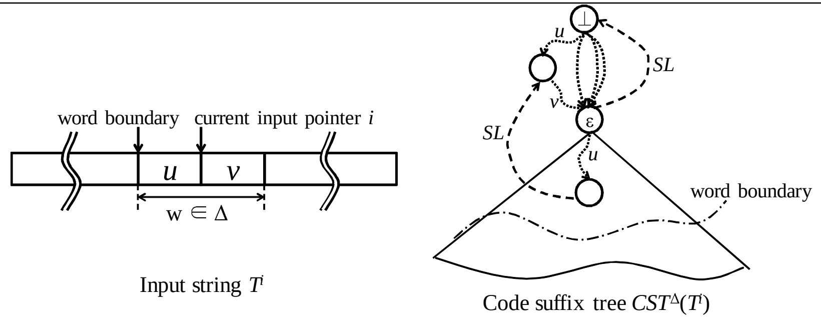

Fig. 5. Proof sketch for the time complexity in the case of cyclic code automata

Lemma 3. *For every stage $i = 1, \ldots, n$, The procedure* Extend *in Fig. 4 correctly computes the border $Bd^\Delta(i)$ from $Bd^\Delta(i-1)$ and $a_i = T[i]$.*

Proof. It is easy to see that Extend correctly implements the extensions of suffixes in $Bd^\Delta(i)$ mentioned above. Then, the most part is shown in a similar way to Ukkonen [17]. For extension of all three types, the above procedure correctly extends the original suffix $[\alpha] \in Bd^\Delta(i-1)$ to obtain $[\alpha a_i] \in Bd^\Delta(i)$. Remaining thing is to show the while-loop from lines 2 to 13 of Extend eventually terminates. If the while-loop is executed repeatedly, the depth of the extension pointer ϕ become smaller, and finally, either it ends with extension of type 3 or it enters the domain dom(code). In the former case, the proof is done. In the latter case, ϕ enters dom(code) and thus ends with extension of type 3, too. This completes the proof. □

From Lemma 2 and Lemma 3, we show the following theorem.

Theorem 1 (correctness). *For every stage $i = 1, \ldots, n$, the algorithm* ConstructCST *in Fig. 3 correctly constructs* $\mathsf{CST}^\Delta(T^i)$*.*

3.3 Time complexity

The remaining task is to estimate the time complexity of ConstructCST in Fig. 3. Let N_{tree} and N_{code} be the numbers of tree and code edges traversed during the computation, respectively. Let $N = N_{tree} + N_{code}$. In a special case that Δ is finite and thus DFA(Δ) is acyclic, the linear time complexity of ConstructCST can be easily proved by applying the *telescope argument* on the changes of the depth $D(\phi)$ of ϕ as used in [17] with a little twist that $D(\phi)$ is defined by the number of the code and tree nodes on the path from $[\perp]$ to ϕ.

In the general case that DFA(Δ) $= A$ is possibly cyclic, and consequently Δ is infinite, however, it is not straightforward to show a linear bound of $N = N_{tree}+N_{code}$ because the extension pointer ϕ can move inside a cycle in dom(tree) many times without monotonically increasing the depth parameter $D(\phi)$, and thus, it is not sufficient to linearly bound N_{code} for dom(code). To overcome this difficulty, we bound the number N_{code} by the total number of letters consumed during the traversal on dom(code). We have the main theorem of this paper.

Theorem 2 (linear time construction of code suffix trees). *Let $\Delta \subseteq \Sigma^*$ be a regular prefix code on Σ recognized by a code automaton $A = \mathsf{DFA}(\Delta)$. Then, the algorithm* ConstructCST *in Fig. 3 constructs $\mathsf{CST}^\Delta(T)$ for an input text T in $O(n \log|\Sigma| + m)$ time and $O(k)$ space in online manner, where $k = |T|_\Delta$, $n = |T|_\Sigma$, and $m = ||A||$ is the size of A.*

Proof. We show that the number $N = N_{tree} + N_{code}$ is bounded by $i = |T^i|$ for every stage i. We can show that the number N_{tree} in $\mathsf{dom}(\text{tree})$ is linearly bounded by $i = |T^i|$. Therefore, we estimate the total number N_{code} of all child and suffix links that the algorithm traverses in $\mathsf{dom}(\text{code})$ through all stages. Let $i = 1, \ldots, n$ be any stage, and let $T^i = T[1] \cdots T[i] = w_1 \cdots w_k u_{k+1} \in \mathsf{Pre}(\Delta^*)$ be the current input string. At stage i, we denote by ∂N^i_{code} the number of suffix and child edge traversals added to N_{code}. Then, there are three cases below when ϕ traverses inside $\mathsf{dom}(\text{code})$: (a) The case that at stage $i-1$, the extension ends at node ϕ in $\mathsf{dom}(\text{code})$ such that $\phi \neq [\varepsilon]$. From the construction of $\mathsf{DFA}\Delta$ and Lemma 3, the algorithm executes exactly one extension of type 3 in $\mathsf{dom}(\text{code})$ by going down a child edge. Thus, $\partial N^i_{code} \leq 1$ is immediate. (b) Otherwise, at stage $i-1$, the extension ends at node $\phi \notin \mathsf{dom}(\text{code})$. This implies that $\phi \in \mathsf{dom}(\text{tree})$. In this case, the algorithm repeats extensions of type 2 in the while-loop from lines 2 to 13 of Extend by traversing suffix links in $Bd^\Delta(i-1)$, and terminates with extension of type 3. Let $\phi = [\beta]$ be the final extension point of type 2. Then, we have two subclasses below on ϕ: (b.1) The case that $\phi \in \mathsf{dom}(\text{tree})$. Since all the preceding extension for $Bd^\Delta(i-1)$ were done in $\mathsf{dom}(\text{tree})$, we have $\partial N^i_{code} = 0$. (b.2) The case that $\phi \notin \mathsf{dom}(\text{tree})$. Then, $\phi = [\beta] \in \mathsf{dom}(\text{code})$ for a preword β. Let $\phi = \langle s, j, i-1 \rangle$ be the canonical pointer of $[\beta]$. From Lemma 3, we can show that $\beta = u_{k+1}$ holds, that is, the string label $\beta = L(\phi)$ coincides the current tail preword u_{k+1} of T^i being scanned. In Extend, we then move from $\phi = [\beta]$ to $SL^\Delta(\beta) = [\perp\beta]$ by firstly following one suffix link from the real node s and by successively going down at most $|\beta|$ child edges by applying Canonize$(SL^\Delta(s), j, i-1)$ at line 11 (See Fig. 5). Therefore, ∂N^i_{code} is at most $|\beta| + 1$. On the other hand, suppose that we are scanning a prefix $\beta = u_{k+1}$ of some complete word w_{k+1}. Then, it is not hard to show that the extension in the case (b.2), where a jump from $\mathsf{dom}(\text{tree})$ into $\mathsf{dom}(\text{code})$ is performed, can occur at most once per complete codeword w_{k+1} during the whole scan, because once the case (b.2) occurs, only the case (b.3) can occurs iteratively until ϕ reaches $\mathsf{dom}(\text{tree})$. Thus, we can amortize the cost for the case (b.2) over the whole computation. Combining the above arguments, we have the number of edge traversals bounded by:

$$N_{code} \leq \sum_{i=0}^{n} \partial N^i_{code} \leq (\sum_{i=0}^{n} 1) + (\sum_{j=0}^{k+1} |w_j|) \leq 2n,$$

where we used the inequality $\sum_{j=0}^{k+1} |w_j| = |T| = n$. From similar arguments as in [17], we can show the remaining part that $N_{tree} \leq 2n$. Hence, the total number of edge traversals is given by $N = N_{tree} + N_{code} \leq 4n$. Space complexity is obvious since $\mathsf{CST}^\Delta(T)$ has at most $O(n)$ real nodes. For the total time complexity,

```
Algorithm Terminate($\psi = \langle s, j, i-1 \rangle, i$):
Input: A terminating point $\psi$ and $i \geq 0$;
$\psi \leftarrow$ Canonize($\langle s, j, i \rangle$); {type 1 extension by letter $a_i$}
ChildTrans($\Theta', T[i]$);
if toClose($\Theta'$) then
    $v := child(s, T[j])$;
    if $label(v) = \langle j, \infty \rangle$ then $label(v) \leftarrow \langle j, i \rangle$;
    SuffixTrans($\Theta', \psi$);
return $\psi$;
```

Fig. 6. Terminating truncated word suffix trees

the algorithm takes $O(m)$ time for the preprocessing $A = \mathsf{DFA}(\Delta)$ takes, and takes and $O(1)$ time per suffix link traversal and $O(\log |\Sigma|)$ time per child edge traversal with an appropriate dictionary structure. Hence, by construction of the main algorithm, the time complexity is $O(n)$. □

For a fixed base alphabet Σ, the algorithm runs in $O(n + m)$ time and $O(k)$ space.

4 Application to Truncated Code Suffix Trees

Let $T \in \mathsf{Pre}(\Delta^*)$ $T = T[1] \cdots T[n] = w_1 \cdots w_k w_{k+1} \in \mathsf{Pre}(\Delta^*)$ be an input prestring on Δ and $\ell > 0$ be a fixed integer. Then, the *ℓ-truncated code suffix tree (ℓ-TCST)* of T on Δ, denoted by $\ell\text{-}\mathsf{TCST}^{\Delta}(T)$, is a compacted trie that represents the set $\mathsf{Fac}^{\Delta}(T, \ell)$ of all (Δ, ℓ)-factors of T. It is easy to see that the number of nodes in $\ell\text{-}\mathsf{TCST}^{\Delta}(T)$ is linear in the number $k' = |\mathsf{Fac}^{\Delta}(T, \ell)|$ of unique (Δ, ℓ)-factors of T. Since k' is smaller than $k = |\mathsf{Suf}^{\Delta}(T)|$ in real data sets, we expect that ℓ-TCST is more space efficient than CST for small values of ℓ.

The modified algorithm ConstructTCST is obtained from the original ConstructCST of Fig. 3 by inserting Terminate in Fig. 6 after Extend of line 8. The main difference of the new algorithm from the old one is the use of the *termination pointer* ψ for closing suffixes in addition to the extension pointer ϕ for opening suffixes. At every stage $i = 1, \ldots, n$, Extend first extends each $\phi = \alpha$ of type 2 in T^{i-1} to αa_i by attaching the i-th letter $a_i = T[i] \in \Sigma$. At the same time, Terminate keeps track of termination point ψ, which is an open leaf $\psi = \langle j, \infty \rangle$ with word depth at most $\ell - 1$, and terminates it whenever ϕ reaches the depth ℓ by replacing $\langle j, \infty \rangle$ with $\langle j, i \rangle$, where i is the current index. A key observation is that there exists at most one open leaf to be closed at every stage i.

To implement this idea, we have to count the length of the open suffixes in the number of codewords to detect when $|L(\psi)|$ exceeds the limit ℓ. To do this we use a data structure $\Theta = \langle \eta, wc \rangle$, where $\Theta.\eta = \eta \in \mathsf{dom}(\mathrm{code})$ is a *boundary pointer* to a code node and $\Theta.wc = wc \in \mathbb{N}$ is a *word counter*, with the following operations:

- $\mathsf{Reset}(\Theta) \equiv$ **begin** $\Theta.\eta \leftarrow [\bot]$; $\Theta.wc \leftarrow 0$; **end**.
- $\mathsf{Decrement}(\Theta) \equiv \Theta.wc \leftarrow \Theta.wc - 1$;
- $\mathsf{ChildTrans}(\Theta, C) \equiv$

```
1: $\Theta.\eta \leftarrow child_{\mathsf{DFA}(\Delta)}(\Theta.\eta, C)$;
2: if $\Theta.\eta = [\varepsilon]$ then begin $\Theta.\eta \leftarrow \bot$; $\Theta.wc \leftarrow \Theta.wc + 1$ end;
```

- $\mathsf{SuffixTrans}(\Theta, \phi = \langle s, j, i\rangle) \equiv$

```
1: $\Theta.wc \leftarrow \Theta.wc - 1$; $\phi \leftarrow \mathsf{Canonize}(\langle SL^{\Delta}(s), j, i\rangle)$; return $\phi$;
```

- $\mathsf{toClose}(\Theta) \equiv$

```
1: return $\Theta.wc = k$ and $\Theta.\eta = [\varepsilon]$;
```

The meaning of the above operations will be easily understood. Using these operations, we modify the algorithm ConstructCST and procedures Extend and Canonize by adding comment lines with "for ℓ-TCST." In Canonize, we replace the sentence "**if** $p - q > i - j$ **then**" at line 4 with "**if** $p - q > i - j$ **or** u is a leaf **then**." For time complexity, ChildTrans takes $O(\log |\Sigma|)$ time, and all the other operations than SuffixTrans takes constant time. By analysis similar to one in the previous section, we can show that SuffixTrans requires amortized constant time per operation. From a similar discussion in Sec. 3 and in Na *et al.* [14], we can show the following theorem.

Theorem 3 (linear time construction of a TCST on prefix code). *If Δ be a fixed, possibly infinite prefix code, then the modified algorithm* ConstructTCST *constructs a truncated code suffix tree ℓ-$\mathsf{TCST}^{\Delta}(T)$ for an input text T in $O(n)$ time and $O(k)$ space in online manner, where $k = |T|_{\Delta}$ and $n = |T|_{\Sigma}$.*

5 Experimental Results

We ran experiments on real datasets. Input data were an English text from the Pizza & Chili Corpus [1], where the delimiters are spaces SPC and LF, and a UTF-8 text from the Mainichi Newspaper Corpus 1991 in Japanese [2]. The length of each text was 50MB. The English text has 336,578 different words of the average length 5.20 (byte). The UTF-8 text has 4054 different codes of the average length 2.96 (byte).

We implemented several types of sparse and truncated suffix trees. ST is the suffix tree, CST is the code suffix tree in Chapter 3, IST is the suffix tree over the code alphabet Δ using four-byte integers as base letters, HST is the code suffix tree over the (letter-based) Huffman code, and IHST is the code suffix tree over the word-based Huffman code for Δ. TST, TCST, TIST, THST and TIHST are their truncated versions with the factor length $\ell = 2, 5$, and 10 (words). These programs were written in C++ and compiled by Microsoft Visual Studio 2010. We ran the programs on an Intel Core i7 920 and 12GB of RAM, running Windows 7 Professional 64bit.

[1] http://pizzachili.dcc.uchile.cl/

[2] http://www.nichigai.co.jp/sales/corpus.html

Table 1. Node count (10^6 nodes), where ℓ is the length of code factors.

Data	ST	CST	IST	HST	IHST	ℓ	TST	TCST	TIST	THST	TIHST
English	87	17	14	105	20	2	28	3.6	2.6	39	4.9
						5	56	11	8.3	74	14
						10	58	11	8.7	76	14
UTF-8	85	29	25	105	35	2	2.5	0.43	0.35	4.0	0.70
						5	43	13	11	58	17
						10	70	23	20	89	30

Table 2. Query time (microseconds per query).

Data	ST	CST	IST	HST	IHST	ℓ	TST	TCST	TIST	THST	TIHST
English	1.03	0.827	0.686	1.295	0.624	2	0.889	0.718	0.577	1.139	0.546
	1.233	0.952	0.827	1.497	0.733	5	1.170	0.905	0.764	1.435	0.718
	1.263	0.952	0.827	1.514	0.749	10	1.185	0.904	0.827	1.467	0.718
UTF-8	0.537	0.47	0.441	0.787	0.434	2	0.421	0.341	0.305	0.608	0.304
	0.88	0.772	0.803	1.138	0.718	5	0.827	0.72	0.684	1.077	0.677
	0.886	0.774	0.767	1.138	0.723	10	0.858	0.755	0.787	1.108	0.705

Tables 1 and 2 show the node counts of the suffix trees and the average query time for 10^6 strings of the code lengths $\ell = 2, 5$, and 10 (words), respectively. In the experiments, we observed that the sparse suffix trees were more space-efficient and faster than their non-sparse versions, roughly, by the factor of $O(n/k)$. Truncated suffix trees also improved space efficiency and query time in both full and truncated versions. IHST was the fastest in the algorithms, even though HST was slowest. CST was slightly slower than IST, however, it is comparable because it does not need any additional space and preprocessing.

6 Conclusion

In this paper, we presented an online construction algorithm for CSTs and ℓ-TCSTs on regular, variable-length prefix codes that runs in $O(n + m)$ time and $O(k)$ space, where n is the total text size, k is the number of codewords, and m is the size of a code automata.

As future works, Extensions of this approach to other suffix indexes, e.g., DAWG [9], CDAWG [10], and suffix arrays [5] and application to enhanced suffix arrays [1, 12] and property suffix trees [2, 16] would be interesting. Also, it would be an interesting future problem to study lowerbounds of the worst case time complexity of construction of sparse suffix trees with an arbitrary index set when $O(k)$ space is allowed.

Acknowledgements

The authors are grateful to anonymous referees for many useful comments and suggestions that significantly improve the quality of this paper. They also would like to thank Thomas Zeugmann, Ayumi Shinohara, Masayuki Takeda, Takuya Kida, Shin-ichi Minato, Makoto Haraguchi, and Osamu Watanabe for their discussions and valuable comments, This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 20240014, FY2008–2011, and MEXT/ JSPS Global COE Program, "*Center for Next-Generation Information Technology based on Knowledge Discovery and Knowledge Federation*," FY2007–2011.

References

1. M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, *J. of Discrete Algorithms*, 2(1), 53–86, 2004.
2. A. Amir, E. Chencinski, C. Iliopoulos, T. Kopelowitz, and H. Zhang, Property machting and weighted matching, *Proc. CPM'06*, LNCS 4009, 188–199, 2006.
3. A. Andersson, N. J. Larsson, and K. Swanson, Suffix trees on words, *Algorithmica*, 23(3), 246–260, 1999.
4. M. Crochemore and W. Rytter, *Jewels of Stringology: Text Algorithms*, 2002.
5. F. Ferragina and J. Fischer, Suffix arrays on words, *Proc. CPM'07*, LNCS 4580, 328–339, 2007.
6. D. Gusfield, *Algorithms on Strings, Trees, and Sequences*, – Computer science and computational biology, Cambridge, 1997.
7. IETF, UTF-8, a transformation format of ISO 10646, RFC 3629, 2003. http://tools.ietf.org/html/rfc3629
8. S. Inenaga and M. Takeda, On-line linear-time construction of word suffix trees, *Proc. CPM'06*, LNCS, Springer, 60–71, 2006.
9. S. Inenaga and M. Takeda, Sparse directed acyclic word graphs, *Proc. SPIRE'06*, LNCS 4209, Springer, 61–73, 2006.
10. S. Inenaga and M. Takeda, Sparse compact directed acyclic word graphs, Jan Holub, Jan Zdarek (Eds.), *Proc. PSC'06*, 197-211, 2006.
11. J. Kärkkäinen and E. Ukkonen, Sparse suffix trees, *Proc. COCOON'96*, LNCS, Springer, 219–230, 1996.
12. T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, *Proc. CPM'01*, LNCS 2089, 181–192, 2001.
13. E. M. McCreight, A space-economical suffix tree construction algorithm, *J. ACM*, 23, 262–272, 1976.
14. J.C. Na, A. Apostolico, C.S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304:87–101, July 2003.
15. M. Takeda *et al.*, Processing text files as is: pattern matching over compressed texts, multi-byte character texts, and semi-structured texts, *Proc. SPIRE'02*, LNCS 2476, 2002.
16. T. Uemura, H. Arimura, A linear-time off-line construction of property suffix trees, *IEICE Trans. Inf. & Syst.*, J91-D(3), 595–607, 2008 (in Japanese). An English version appears in Chapter 4, T. Uemura, *Efficient Construction of Constrained Suffix Trees*, Ph.D thesis, IST, Hokkaido Univ., Feb. 2011 (submitting).
17. E. Ukkonen, On-line construction of suffix-trees, *Algorithmica*, 14(3), 249-260, 1995.