Efficient Construction of
Constrained Suffix Trees
（制約付き接尾辞木の効率良い構築）

Takashi Uemura

February, 2011
Abstract

Suffix tree, proposed by Weiner in 1973, is a well-known full-text index structure that stores all the substrings in a given text. For a text $T$ of length $n$, the suffix tree for $T$ can be constructed in $O(n \log |\Sigma|)$ time and $O(n)$ space, where $T$ is a string over an alphabet $\Sigma$ and $|\Sigma|$ is the size of $\Sigma$.

However, due to the rise of the large-scale and semi-structured text data, several applications need more powerful and flexible index structures.

For this reason, we consider a constraint for the text and then extend the suffix tree to store all the substrings in the text that satisfy the constraint. Applying a constraint is not only effective for reducing the space consumption of the constrained index structure, but also for improving the computation time for searching queries, because the number of the results for queries will get fewer than the number of the results by the normal suffix tree.

In this thesis, we propose a family of the constrained suffix trees, which are suffix trees that store the set of all substrings in an input text that satisfy a given constraint.

In Chapter 3, we propose the word-based truncated suffix tree and its efficient construction algorithm. We here assume that the input text is written in a natural language such as English and each word is splitted by some special character like the white space. Given an integer $k > 0$ and a text $T$, the $k$-word-based suffix tree for $T$ is an index structure that stores any substring which consists of at most $k$ words. Then, we show an online construction algorithm that runs in $O(n \log |\Sigma|)$ time.
In Chapter 4, we study efficient construction of property suffix trees, which is an index structure proposed by Amir et al. in 2006. Property is a set of intervals over a text. Given a text $T$ with a property $\pi$, the property suffix tree for $T$ and $\pi$ is an index structure that stores any substring in $T$ which is included by an interval in $\pi$. Amir et al. presented a $O(n \log |\Sigma| + n \log \log n)$ time construction algorithm. In this chapter, we focus attention on a subproblem that computing the borders between the outsides and the insides of the intervals. We then give an efficient algorithm for solving the subproblem and show that the property suffix tree can be constructed in $O(n \log |\Sigma|)$ time.

In Chapter 5, we propose the word $N$-gram tree, which is an extension of the word-based truncated suffix tree. The word-based truncated suffix tree, proposed in Chapter 3, indexes not only all the sequences of at most $k$ words but also their substrings. It is not a desirable for some applications. On the other hand, there is another word-based index structure, called the word suffix tree, proposed by Anderson et al. in 1999 and showed its online construction algorithm by Inenaga et al. in 2006. Given a text $T$, the word suffix tree stores any substring which starts at a head of a word in $T$. In other words, the constraint considered in the word suffix tree limits the start positions of the substrings, but do not limit the end positions. The word $N$-gram tree is therefore an extension of both the word-based truncated suffix tree and the word suffix tree. In this chapter, we also consider a keyword extraction problem for Web browsing using the word $N$-gram trees. In information gathering from Web pages, there are various candidates for keywords, such as titles of books or movies, slang terms, and newly-coined expressions. We propose a keyword extraction algorithm using word $N$-gram trees and discuss the keyword extraction from a book and blogs.

In Chapter 6, we study an unsupervised spam document detection problem using suffix trees. Spam document is a message sended to the general public for the purpose of advertisements. Given a document $d$ in a set of documents $D$, we consider the
occurrence probability $P(d|D)$ of $d$ given a probabilistic model derived from $D$. Then, we regard $d$ as a spam document if $P(d|D)$ is unnaturally high. This strategy is based on the assumption that the spam creator generates a number of similar spam documents and then their occurrence probabilities become high if there are many similar documents in $D$. To compute the occurrence probability, we present an algorithm which computes all the occurrence probabilities for the documents in $D$ in $n \log |\Sigma|$ time, where $n$ is the total length of the documents in $D$ and $\Sigma$ is the alphabet size. In comparison with the methods proposed by Narisawa et al. in 2007, our method works well especially for the spam documents called word salads, which are created by replacements with advertising keywords.

Finally, in Chapter 7, we propose a method for improving the compression ratio of a suffix tree-based data compression method. VF (variable-length-to-fixed-length) coding is a kind of data compression method that constructs a parse tree, which is a representation of a dictionary of substrings, and then parse the text into a sequence of elements represented in the parse tree. Since each elements in the parse tree is associated a unique code of a fixed length, it may be important that choose a long and a frequent substring as an element to obtain a good compression ratio. STVF code, which is a kind of VF codes, constructs the parse tree by choosing substrings from the suffix tree for the input text based on their frequencies. In this chapter, we propose a method that firstly compresses the text by using the parse tree and then reconstructs the parse tree by replacing unnecessary substrings with new candidates derived by the compression. Applying the method repeatedly to the parse tree, we can obtain a better compression ratio than the initial parse tree. Experimental results showed that STVF code with the training approach for the parse tree obtained compression ratios between gzip and bzip2, which are well-known compression programs.
## Contents

1 Introduction 1

2 Preliminaries 7
   2.1 Basic definitions 7
   2.2 Suffix trees 8
   2.3 Online construction of suffix trees 11

3 Online linear-time construction of word-based truncated suffix trees 19
   3.1 Word-based truncated suffix trees 19
   3.2 Construction algorithm 21
   3.3 Analysis 26
   3.4 Counting frequencies 29
   3.5 Experimental results 29
   3.6 Discussion 30

4 Offline linear-time construction of property suffix trees 33
   4.1 Text with property 33
   4.2 Property suffix trees 35
   4.3 Proposed algorithm 43
   4.4 Analysis 47
   4.5 Experimental results 53
5 Keyword extraction and browsing support 59
  5.1 Keyword Extraction Problem .................................. 60
  5.2 Truncated Word Suffix Trees .................................. 63
  5.3 Proposed Method ................................................. 65
  5.4 Experimental results ............................................. 76
  5.5 Discussion ........................................................ 81

6 Unsupervised spam detection by document complexity estimation 83
  6.1 Preliminaries ...................................................... 86
  6.2 Proposed Method .................................................. 89
  6.3 Experimental Results .............................................. 97
  6.4 Discussion ........................................................ 102

7 Training parse trees for efficient variable-to-fixed-length coding 111
  7.1 Variable-Length-to-Fixed-Length Codes ......................... 113
  7.2 STVF Codes ....................................................... 115
  7.3 Proposed Method .................................................. 118
  7.4 Experimental Results .............................................. 120
  7.5 Discussion ........................................................ 122

8 Conclusion .......................................................... 131
  8.1 Summary of the Results .......................................... 131
  8.2 Future Researches ................................................ 132
Chapter 1

Introduction

The rapid growth of the internet has been changing the world. In the theoretical viewpoint, it is supported by a number of efficient algorithms. Text processing is particularly important field for applications such as Web search engines and biological sequence analysis. In addition, it is also an important basis for another kind of algorithms.

Suffix tree, proposed by Weiner [55], is a well-known full-text index structure that stores all the substrings in a given text. For a text $T$ of length $n$, the suffix tree for $T$ can be constructed in $O(n \log |\Sigma|)$ time and $O(n)$ space, where $T$ is a string over an alphabet $\Sigma$ and $|\Sigma|$ is the size of $\Sigma$. Given another string $P$ of length $m$, suffix tree can find all the occurrences of $P$ in $T$ in $O(m + occ)$ time, where $occ$ is the number of occurrences. An online $O(n \log |\Sigma|)$ time suffix tree construction algorithm is presented by Ukkonen [54].

However, due to the rise of the large-scale and semi-structured text data, several applications need more powerful data structures.

For this reason, we consider a constraint for the text and then extend the suffix tree to store all the substrings in the text that satisfy the constraint. For example, consider a constraint that “any substring that appears more than ten times in the text.” Then,
the problem is how to construct the index structure that stores the such substrings. Applying a constraint is not only effective for reducing the space consumption of the constrained index structure, but also for improving the computation time for searching queries, because the number of the results for queries will get fewer than the number of the results by the normal suffix tree.

In this thesis, we propose a family of the constrained suffix trees, which are suffix trees that store the set of all substrings in an input text that satisfy a given constraint.

A trivial way to construct the constrained suffix trees is to enumerate all the substrings that satisfy the constraint and then add them into the tree structure. However, this approach will not work efficiently because there are a quadratic number of substrings in the input text. In addition, it will also take a quadratic space in the worst case. Therefore, we have to develop efficient construction algorithms of constrained suffix trees for large-scale text processing applications.

There are certain variants of the suffix tree. Larsson[32] proposed suffix trees with a sliding window, which indexes all the substrings in the sliding window. Assuming a fixed size alphabet, it takes $O(W)$ space and $O(n)$ time, where $W$ is the length of the sliding window and $n$ is the total length of the whole input text. Na et al.[36] proposed a suffix tree-based index structure which only stores the substrings in an input text of a fixed length. Andersson et al.[3] proposed the word suffix trees which indexes the substrings which begins at the first letters of words. Given an input text $T$ of length $n$ which consists of $m$ words, the word suffix tree for $T$ takes $O(m)$ space except for $T$ itself. Inenaga et al.[21] gave an online linear-time construction algorithm for word suffix trees.

In Chapter 3, we propose the word-based truncated suffix tree and its efficient construction algorithm. We here assume that the input text is written in a natural language such as English and each word is splitted by some special character, called delimiter. Given an integer $k > 0$ and a text $T$, the $k$-word-based suffix tree for $T$ is
an index structure that stores any substring which consists of at most $k$ words. Then, we show an online construction algorithm that runs in $O(n \log |\Sigma|)$ time.

In Chapter 4, we study efficient construction of property suffix trees, which is an index structure proposed by Amir et al. [1]. Property is a set of intervals over a text. Given a text $T$ with a property $\pi$, the property suffix tree for $T$ and $\pi$ is an index structure that stores any substring in $T$ which is included by an interval in $\pi$. Amir et al. presented a $O(n \log |\Sigma| + n \log \log n)$ time construction algorithm. In this chapter, we focus attention on a subproblem that computing the borders between the outsides and the insides of the intervals. The algorithm in et al. [1] utilizes a data structure for solving the weighted ancestor problem [14, 2] and then their algorithm runs in $O(n \log \log n)$ time. To improve their result, we give an efficient algorithm for solving the subproblem and show that the property suffix tree can be constructed in $O(n \log |\Sigma|)$ time.

In Chapter 5, we propose the word $N$-gram tree, which is an extension of the word-based truncated suffix tree. The word-based truncated suffix tree, proposed in Chapter 3, indexes not only all the sequences of at most $k$ words but also their substrings. It is not a desirable for some applications. On the other hand, there is another word-based index structure, called the word suffix tree, proposed by Anderson et al. [3] and showed its online construction algorithm by Inenaga et al. [21]. Given a text $T$, the word suffix tree stores any substring which starts at a head of a word in $T$. In other words, the constraint considered in the word suffix tree limits the start positions of the substrings, but do not limit the end positions. The word $N$-gram tree is therefore an extension of both the word-based truncated suffix tree and the word suffix tree. In this chapter, we also consider a keyword extraction problem for Web browsing using the word $N$-gram trees. In information gathering from Web pages, there are various candidates for keywords, such as titles of books or movies, slang terms, and newly-coined expressions. We propose a keyword extraction algorithm using word $N$-gram trees and discuss the
keyword extraction from a book and blogs.

In Chapter 6, we study an unsupervised spam document detection problem using suffix trees. Spam document is a message sent to the general public for the purpose of advertisements. Given a document \( d \) in a set of documents \( D \), we consider the occurrence probability \( P(d|D) \) of \( d \) given a probabilistic model derived from \( D \). Then, we regard \( d \) as a spam document if \( P(d|D) \) is unnaturally high. This strategy is based on the assumption that the spam creator generates a number of similar spam documents and then their occurrence probabilities become high if there are many similar documents in \( D \). To compute the occurrence probability, we present an algorithm which computes all the occurrence probabilities for the documents in \( D \) in \( n \log |\Sigma| \) time, where \( n \) is the total length of the documents in \( D \) and \( \Sigma \) is the alphabet size. In comparison with the methods proposed by Narisawa et al. [37], our method works well especially for the spam documents called word salads, which are created by replacements with advertising keywords.

Finally, in Chapter 7, we propose a method for improving the compression ratio of a suffix tree-based data compression method. VF (variable-length-to-fixed-length) coding is a kind of data compression method that constructs a parse tree, which is a representation of a dictionary of substrings, and then parse the text into a sequence of elements represented in the parse tree. Since each elements in the parse tree is associated a unique code of a fixed length, it may be important that choose a long and a frequent substring as an element to obtain a good compression ratio. STVF code [24, 28], which is a kind of VF codes, constructs the parse tree by choosing substrings from the suffix tree for the input text based on their frequencies. In this chapter, we propose a method that firstly compresses the text by using the parse tree and then reconstructs the parse tree by replacing unuseful substrings with new candidates derived by the compression. Applying the method repeatedly to the parse tree, we can obtain a better compression ratio than the initial parse tree. Experimental results showed that
STVF code with the training approach for the parse tree obtained compression ratios between gzip and bzip2, which are well-known compression programs.
Chapter 2

Preliminaries

This chapter provides basic definitions that to be used in the later chapters and then introduces the suffix tree which is the main data structure we study in this thesis.

2.1 Basic definitions

Let \( \Sigma \) be a finite set of characters. We denote the size of \( \Sigma \) by \( |\Sigma| \). String \( T \in \Sigma^* \) is a sequence of characters over \( \Sigma \). We denote the \( i \)-th character of \( T \) by \( T[i] \). The length of a string \( T = t_1 \ldots t_n \) is denoted by \( |T| = n \). The concatenation of two strings \( S = s_1 \ldots s_m \) and \( T = t_1 \ldots t_n \) is denoted by \( S \cdot T = s_1 \ldots s_m \cdot t_1 \ldots t_n \). The empty string \( \varepsilon \) is the string with length 0. For any string \( S \in \Sigma^* \), \( \varepsilon \cdot S = S \cdot \varepsilon = S \). We denote the set \( \Sigma^* \setminus \{\varepsilon\} \) by \( \Sigma^+ \).

Let \( S = s_1 \ldots s_n \in \Sigma^* \) be a string. Strings \( x, y, \) and \( z \) are a prefix, a substring, and a suffix of \( S \) if \( S = xyz \), respectively. An interval of \( T \) is a pair of integers \( (i, j) \), where \( 1 \leq i, j \leq n \). The substring of \( S \) that starts at \( i \) and ends at \( j \) is denoted by \( S[i \ldots j] \). That is, \( S[i \ldots j] = s_i \ldots s_j \). We define \( S[i \ldots j] = \varepsilon \) if \( i < j \). We denote the set \( \{S[i \ldots j] \mid 1 \leq i, j \leq |S|\} \) of all the substrings of \( S \) by \( \text{Fac}(S) \). We also denote the set \( \{S[1 \ldots j] \mid 0 \leq j \leq |S|\} \) of all the prefixes of \( S \) and the set \( \{S[i \ldots |S|] \mid 1 \leq i \leq |S|+1\} \).
of all the suffixes of $S$ by $\text{Pre}(S)$ and $\text{Suf}(S)$, respectively.

Let $T, P \in \Sigma^*$ be strings. We say that $P$ occurs in $T$ at the position $i$ if $P = T[i \ldots |P|]$, where $1 \leq i \leq |T| - |P|$. The frequency of $P$ in $T$, denoted by $f(T, P)$, is the number of positions where $P$ occur. We define $f(T, \varepsilon) = |T|$ for any string $T$. In text processing, $T$ and $P$ are often called a text and a query, respectively.

A word $w$ is a string $w \in \{W \cdot \# | W \in \Sigma^+\}$, where $\#$ is the delimiter which is not in $\Sigma$. In English texts, $\#$ corresponds to the white space. Phrase is a sequence of words. The word count of a phrase $P = w_1 \cdots w_m \in W^*$, denoted by $|P|_\#$, is the number of $\#$ in $P$. That is, $|S|_\# = m$. In a similar way, the word count of any string $T$, denoted by $|T|_\#$, is also the number of $\#$ in $T$.

### 2.2 Suffix trees

Let $T$ be a string of length $n > 0$. We assume that the last character $T[n]$ does not appear in $T[1 \ldots n - 1]$. The suffix tree of $T$, denoted by $ST(T)$, is a rooted tree [12] $ST(T) = \{V, E, \text{root}, \bot, \text{child}, \text{suf}\}$ that satisfies the following conditions:

- $V$ is a set of nodes.
- $\text{root} \in V$ is the root of $ST(T)$.
- $E$ is a set of edges.
- $\bot$ is a special node $\bot \not\in V$ which is the parent of $\text{root}$.

For nodes $u, v \in V \cup \{\bot\}$, $u$ is the parent of $v$ and $v$ is a child of $u$ if $(u, v) \in E$. For nodes $v, u \in V$, $u$ is an ancestor of $v$ and $v$ is a descendant of $u$ if $u$ is in the path from $\text{root}$ to $v$. Any node $v \in V$ is a descendant and also an ancestor by itself. Leaf is a node which has no children. Inversely, internal node is a node which has at least one child. In suffix trees, any internal node has at least two children. All the out-going edges
leaving from a node always have *labels* which start with mutually different characters. We denote the corresponding label of the ingoing edge of a node $v$ by $\text{label}_v$. Any label are represented by a pair of positions $(i, j)$, where $\text{label}(v) = T[i..j](1 \leq i, j \leq |T|)$ for each node $v \in V$. Each node $v \in V$ represents the unique substring $\langle v \rangle \in \text{Sub}(D)$, where $\langle v \rangle$ is the concatenation of the labels on the unique path from the root to $v$. We denote the node which represents a string $P$ by $\overline{P}$. If such a node does not exist, $\overline{P}$ is undefined.

Given a node $u \in V$ and a character $c$, the function $\text{child} : (V, \Sigma) \to V$ returns the node $v$ which is a child of $u$ and has the label $\text{label}(v)$ starts with $c$. If such a child does not exist, $\text{child}(u, c)$ is undefined. We define that $\text{child}(\bot, c) = \text{root}$ for any character $c \in \Sigma$.

Given a node $u \in V$, the function $\text{suf} : V \to V \cup \{\bot\}$ returns the node $v$ such that $\langle u \rangle = c \cdot \langle v \rangle$, where $c$ is the first character of $\langle u \rangle$. We define that $\text{suf}(\text{root}) = \bot$. For any real node $u \in V$, $\text{suf}(u)$ always exists. We call $\text{suf}(u)$ the *suffix link of $u$*.

Since the length of each label must be more than one, some substrings in $T$ do not represented by the nodes in $V$. We say that a substring $s \in \text{Fac}(T)$ belongs to a node $u \in V$ if $s$ is a proper prefix of $\langle v \rangle$ and no ancestor of $v$ is not such a node.

We consider *virtual nodes*, which are not in $V$, but they represent substrings in $\text{Fac}(T)$ belonging to some nodes in $V$. Let $u, v \in V$ be nodes. We assume that $|\text{label}(e)| > 1$, where $e = (u, v) \in E$. Then, there exists a substring $s$ which belongs to $v$. The *virtual node* on the edge $e$ is the position in $\text{ST}(T)$ which corresponds to $s$. To distinguish from virtual nodes, we call the nodes in $V$ the *real nodes*. For a virtual node $w$ and its corresponding substring $s$, we define $\langle w \rangle = s$ and $\overline{s} = w$.

Figure 2.1 shows the suffix tree $\text{ST}(T)$ for $T = \text{vivid}$$. The white circles are the real nodes and the black nodes are the virtual nodes. The solid lines are the edges and the broken lines are the suffix links. For each leaf $v$, the number beside $v$ is the position of the corresponding suffix $T[i \ldots n]$. 

9
Figure 2.1: The suffix tree for string $T = \text{vivid}$.
Let $w$ be any virtual node which represents a substring $\langle w \rangle = T[k \ldots i]$ of $T$. Then, since $w$ has an ancestor $u \in V$, which is a real node, there exists an integer $j(k \leq j \leq i)$ so that $\langle w \rangle = T[k \ldots i] = \langle u \rangle \cdot T[j \ldots i]$. Therefore any virtual node $w$ can be represented by a triplet $(s, j, i)$, where $s$ is a real node, $j, i$ are integers. Similarly, we can represent any real node $s$ by a triplet $(s, i, i + 1)$, where $1 \leq i \leq |T|$. We call the triplet $\phi = (s, j, i)$ by a reference of the substring $\langle s \rangle \cdot T[j \ldots i]$. The node corresponding to $\phi$ is represented by $\phi$ and the substring corresponding to $\phi$ is represented by $\langle \phi \rangle$.

2.3 Online construction of suffix trees

In this section, we introduce an online algorithm for constructing suffix trees proposed by Ukkonen[54].

Implicit suffix trees

Figure 2.2 shows Ukkonen’s suffix tree construction algorithm[54]. Reading each character $T[i]$ for $1 \leq i \leq n$, the algorithm constructs the suffix tree $ST(T[1 \ldots i])$ by updating the previous suffix tree $ST(T[1 \ldots i - 1])$. In the construction, some suffixes of $T[1 \ldots i]$ may not be represented by leaves. However, after updating with the last character $\$ \in \Sigma$, all the substrings will be represented by leaves. We call such transient trees implicit suffix trees.

Auxiliary procedures

Before we describe the details of the construction algorithm, we introduce three auxiliary procedures which will be used in the main construction algorithm.

A reference $\phi = (s, j, i)$ is canonical if $s$ is the nearest real ancestor of $\phi$. The procedure canonize shown in Fig. 2.3 transforms any reference $\phi = (s, j, i)$ to the
canonical one. This procedure runs in a linear time to the number of the number of the real nodes that it traverses.

Figure 2.5 shows the procedure split which creates the real node $v$ which corresponds to $\langle \phi \rangle$ and return $v$ if $\overline{\phi}$ is a virtual node. Otherwise, it just returns the real node $\overline{\phi}$. The input argument $\phi$ must be canonical because the edge $e$ to be splited connects with the nearest real ancestor of $\overline{\phi}$.

The procedure test shown in Fig.2.4 reports whether or not there exists a node corresponding to the string $\langle \phi \rangle \cdot c$, where $\phi$ is a canonical reference and $c$ is a character in $\Sigma$. It returns $\text{child}(s,c)$ if $\phi$ corresponds to a real node. Otherwise, it reports that whether $c$ equals to the next character in $\text{label}(\text{child}(s,T[i]))$.

**Updating suffix trees**

Now, we describe the procedure update, shown in 2.6. For each position $1 \leq i \leq n$, it appends all the substrings which are not represented in the previous suffix tree $ST(T[1 \ldots i - 1])$.

Let $X_i$ be the longest suffix of $T$ which occurs at least twice in $T[1 \ldots i - 1]$. Then, the $i$-th creating point is the reference $\phi_i$ which corresponds to $X_i$. Any nodes are created at the position of $\phi_i$ or are created as a child of $\phi_i$. We define the initial creating point by $\phi_1 = (\text{root}, 1, 0)$. For each iteration, $\phi_i$ must be canonical.

The procedure appends new leaves to the current suffix tree $ST(T[1 \ldots i - 1])$ which are necessary for $ST(T[1 \ldots i])$ as follows: It creates a new leaf $v$, which represents $\langle \phi \rangle \cdot T[i]$, if $\langle \phi \rangle \cdot T[i]$ is not represented by a node in the current tree. If $\overline{\phi}$ is a virtual node, it also creates the real one and the suffix link. Then $\overline{\phi}$ moves to the node which represents the one fewer suffix of $\langle \phi \rangle$ by using the suffix link of $s$.

The label of each leaf $v$ is set to $\text{label}(v) = (i, \ast)$. In $i$-th iteration, the special symbol $\ast$ is interpreted as $i$. That is, $v$ will represent a suffix of $T[i + 1]$ in the next iteration without any modification of the label. The iteration stops if $\langle \phi \rangle \cdot T[i]$ is found.
Let \( S_i \) be the suffix of \( T[1…i] \) which is represented by \( ⟨ϕ⟩ \cdot T[i] \) after \( i \)-th iteration. All the suffixes which are longer than \( S_i \) are represented by leaves. \( ⟨ϕ⟩ \cdot T[i] \) is represented by \( ST(T[1…i-1]) \) and is a prefix of \( T[1…i] \). Therefore \( ϕ_{i+1} \) is the canonical reference such that \( ⟨ϕ_{i+1}) = ⟨ϕ⟩ \cdot T[i] \). Since \( ⟨ϕ⟩ = ⟨s⟩ \cdot T[j…i-1] \), \( ϕ_{i+1} \) can be obtained by canonize the reference \( (s, j, i + 1) \).

By an amortized analysis, we can show that the construction of the suffix tree can be done in \( O(M) \) time, where \( M \) is the number of created nodes. Finally, updating the implicit suffix tree \( ST(T[1…n]) \) with the terminal character $, it becomes the correct suffix tree.

**Theorem 1 (Ukkonen [54])** Given a string \( T \) of length \( n \), the procedure \texttt{construct-ST} shown in Fig. 2.2 constructs the suffix tree \( ST(T) \) in \( O(n) \) time.
Procedure construct-\(ST\);

**Input**: A string \(T[1\ldots n]\);

**Output**: The suffix tree \(ST(T)\);

1. create the nodes \(root\) and \(\bot\);
2. \(suf(root) := \bot\);
3. for each \(c \in \Sigma\)
   4. create the edge \(e = (root, \bot)\) and the label \(label(e) = c\);
5. \(\phi := (root, 1, 0)\);
6. for \(i = 1\ldots n\)
7. \(\phi := \text{update}(T, ST, i, \phi)\);
8. return \(ST\);

---

Figure 2.2: The main procedure for construction of suffix trees.
Procedure canonize;

Input: A reference $\phi = (s, j, i)$;

Output: The canonical reference $\phi$;

1. if $(j > i)$
2. return $(s, j, i)$;
3. $t := child(s, T[j])$;
4. while ($|label(t)| \leq i - j + 1$)
5. $(s, j, i) := (t, j + |label(t)|, i)$;
6. $t := child(s, T[j])$;
7. }
8. return $(s, j, i)$;

Figure 2.3: A procedure which transforms references into canonical ones.
Procedure test;

Input: A canonical reference $\phi = (s, j, i)$ and a character $c$;

Output: Whether there exists the node that represents $\langle \phi \rangle \cdot c$;

1. if $(j > i)$
2.   if $\text{child}(s, c)$ is defined
3.     return true;
4.   else
5.     return false;
6. else {
7.     $t := \text{child}(s, T[j])$, $(k, l) := \text{label}(t)$;
8.     return $l - k > i - j$ and $T[k + i - j + 1] = c$;
9. }

Figure 2.4: The procedure that tries to traverse the suffix tree.
Procedure split;

Input: A canonical reference \( \phi = (s, j, i) \);

Output: The real node \( t = \bar{\phi} \);

1. \( \text{if } j > i \)
2. \( \text{return } s; \)
3. \( \text{else } \{
4. \quad u := \text{child}(s, T[j]), (k, l) = \text{label}(u);
5. \quad \text{create a node } t, \text{ edges } (s, t), (t, u);
6. \quad \text{label}(t) := (k, k + i - j), \text{label}(u) := (k + i - j + 1, l);
7. \quad \text{delete the edge } (s, u);
8. \quad \text{return } t;
9. \} \)

Figure 2.5: The procedure which creates a new real node corresponding to a virtual node.
Procedure update;

Input: A string $T$, an integer $i$, the suffix tree $ST(T[1 \ldots i - 1])$, and a reference $\phi_i = (s, j, i - 1)$;

Output: The suffix tree $ST(T[1 \ldots i])$ and the reference $\phi_{i+1}$;

1. $t := \text{NIL}$;
2. while test($\phi, T[i]$) = $\text{false}$ {
3. $u := \text{split}(\phi)$;
4. if $t \neq \text{NIL}$
5. $suf(t) := u$;
6. $t := u$;
7. create a node $v$, an edge $(u, v)$, and a label $\text{label}(v) = (i, *)$;
8. $\phi := \text{canonize}((suf(s), j, i - 1))$;
9. }
10. if $t \neq \text{NIL}$
11. $suf(t) := u$;
12. $\phi := \text{canonize}((s, j, i))$;
13. return $ST, \phi$;

Figure 2.6: The updating procedure for suffix trees.
Chapter 3

Online linear-time construction of word-based truncated suffix trees

In this chapter, we propose the word-based truncated suffix tree and its efficient construction algorithm. We here assume that the input text is written in a natural language such as English and each word is split by some special character, called delimiter. Given an integer \( k > 0 \) and a text \( T \), the \( k \)-word-based suffix tree for \( T \) is an index structure that stores any substring which consists of at most \( k \) words. Then, we show an online construction algorithm that runs in \( O(n \log |\Sigma|) \) time.

This chapter is based on the paper [48].

3.1 Word-based truncated suffix trees

Let \( k > 0 \) be an integer and \( T \) be a sequence of length \( n \) with \( m \) words, that is, \( T = w_1 \cdots w_m \). The \( k \)-word-based truncated suffix tree of \( T \), denoted by \( k\text{-WST}(T) \), is a path-compact trie which represents the set of substrings in \( T \ Fac(w_i\#w_{i+1}\#\ldots\#w_{i+k-1}) \) for all \( 1 \leq i \leq n - k + 1 \).

More formally, \( k\text{-WST}(T) \) is a rooted tree \( \{V, E, root, \perp, child, suf\} \) that satisfies
the following conditions:

- $V$ is a set of nodes.
- $\text{root} \in V$ is the root of $k$-$WST(T)$.
- $E$ is a set of edges.
- $\bot$ is a special node $\bot \not\in V$ which is the parent of $\text{root}$.

Any internal node has at least two children. All the out-going edges leaving from a vertex always have labels which start with mutually different characters. We denote the corresponding label of the ingoing edge of a node $v$ by $\text{label}_v$. Any label are represented by a pair of positions $(i, j)$, where $\text{label}_v = T[i..j](1 \leq i, j \leq |T|)$ for each node $v \in V$.

Each vertex $v \in V$ represents a unique substring $\langle v \rangle \in \bigcup_{1 \leq i \leq |T| - N + 1} \text{Fac}(w_i \# w_{i+1} \# \cdots \# w_{i+k-1})$, where $\langle v \rangle$ is the concatenation of the labels on the unique path from the root to $v$.

Any leaf represents a substring of $k$ words or a suffix of $T$. We denote the node which represents a string $P$ by $\overline{P}$. If such node does not exist, $\overline{P}$ is undefined.

As an example, Fig. 3.1 shows the 2-word-based suffix tree for a text $T = \text{"this} \# \text{is} \# \text{the} \# \text{pen"}$ and Fig. 3.2 shows the substrings represented by 2-$WST(T)$.
In the rest of this chapter, we will propose a construction algorithm of $k$-$WST(T)$ and then show that the algorithm constructs $k$-$WST(T)$ in $O(n)$ time and space for a fixed alphabet. That is, the main result of this chapter is as follows:

**Theorem 2** Given an integer $k > 0$ and a sequence of $m$ words $T$ of length $n$, $k$-$WST(T)$ can be constructed in $O(n)$ time and space.

### 3.2 Construction algorithm

Figure 3.5 shows the proposed algorithm construct-$k$-$WST$. This algorithm consists of two phases: the expanding phase and the terminating phase.
Procedure construct-$k$-WST;

**Input**: a text $T[1 \ldots n]$, an integer $k$;

**Output**: The word based truncated suffix tree $k$-WST($T$);

1. create $root$ and $\bot$ for $k$-WST($T$);
2. $suf(root) := \bot$;
3. **for each** $c \in \Sigma$
   4. create an edge $e = (root, \bot)$ and a label $\text{label}(e) = c$;
5. $\phi := (root, 1, 0)$;
6. $num_\phi = 0$;
7. $start_\phi = 0$;
8. $\psi := (root, 1, 0)$;
9. $num_\psi = 0$;
10. $start_\psi = 0$;
11. **for** $i = 1 \ldots n$
12. \hspace{1cm} $(\phi, num_\phi, lpos_\phi, i) := \text{create}(\phi, num_\phi, lpos_\phi, i)$;
13. \hspace{1cm} $(\psi, num_\psi, lpos_\psi, i) := \text{terminate}(\psi, num_\psi, lpos_\psi, i)$;
14. }
15. return $k$-WST($T$);

Figure 3.3: Construction algorithm for $k$-word-based suffix trees.
Expanding phase

Figure 3.4 shows the procedure `create` which expands word-based truncated suffix trees. This procedure is based on `update` in Section 2.3. The main difference between `update` is that it stores the word count $num_\phi = |\langle \phi \rangle|_#$ for the reference $\phi$ and keeps $\phi$ representing at most $k$ words.

The maintenance of $num_\phi$ is follows: increment if $\phi$ follows an edge with the delimiter `#` and decrement if $\phi$ goes to its suffixes and the first character of $\langle \phi \rangle$ is `#`. For the latter part, we adopt a variable `start_\phi` which stores the start position of $\langle \phi \rangle$. That is, $start_\phi = j - |\langle s \rangle|$, where $\phi = (s, j, i)$. The variable `start_\phi` starts from 1 and is incremented if $\phi$ follows a suffix link.

For any $1 \leq i \leq n$, $\phi_i$ denotes the canonical reference which is to be used in the $i$-th update in the construction of $k$-WST$(T[1\ldots i])$. Then, the string $\langle \phi_i \rangle$ is the longest substring which is a suffix of $T[1\ldots i-1]$ consists of at most $k$ words and occurs at least twice in $T[1\ldots i-1]$.

Terminating phase

In suffix trees, the end position of the label for any leaf is `*`. Hence, once we create a leaf, we do not need to maintain the leaf. However, in online construction of word-based truncated suffix trees, any leaf should represent a substring consists of $k$ words. The procedure `terminate` shown in Fig. 3.5 fixes

Let $\psi$ be the canonical reference such that $\langle \psi \rangle$ is the longest suffix represented by $k$-WST$(T)$. For any point $1 \leq i \leq n$ of the construction, the procedure `terminate` checks the leaf $v = \overline{\psi}$ and overwrite `*` with the end position $i$. We call this operation the closing of $v$. If a leaf is closed, we may also close some other leaves. Moving $\psi$ by using suffix links, we close all the leaves which represent strings consist of $k$ words. Then, $\psi$ reaches a leaf which represents a string consists of $k - 1$ words. This $\psi$ is the
Procedure create;

Input: integers $i, k$, a text $T[1 \ldots i]$, a $k$-word-based suffix tree $k$-WST($T[1 \ldots i-1]$), a reference $\phi_i = (s,j,i-1)$, the word count $num_\phi$ of $\phi$, and the start position $\text{start}_\phi$ of $\langle \phi_i \rangle$;

Output: A word based truncated suffix tree $k$-WST($T$), the reference $\phi_{i+1}$, the word count $num_{\phi}$ of $\phi_{i+1}$, and the start position $\text{start}_{\phi}$ of $\langle \phi_{i+1} \rangle$;

1 $t := \text{NIL}$;  
2 while test($\phi, T[i]$) = false {
3 $u := \text{split}(\phi)$;  
4 if $t \neq \text{NIL}$
5 $\text{suf}(t) := u$;  
6 $t := u$;  
7 create a node $v$, an edge $(u, v)$, and a label $\text{label}(v) = (i, *)$;  
8 $\phi := \text{canonize}((\text{suf}(s), j, i - 1))$;
9 }  
10 if $t \neq \text{NIL}$
11 $\text{suf}(t) := u$;  
12 $\phi := \text{canonize}((s, j, i))$;  
13 if $T[i] = \#$
14 $num_\phi = num_\phi + 1$;  
15 while $num_\phi = k$
16 $\phi := \text{canonize}((\text{suf}(s), j, i))$;  
17 if $T[\text{start}_\phi] = \#$
18 $num_\phi = num_\phi - 1$;  
19 $\text{start}_\phi = \text{start}_\phi + 1$;
20 }
21 }
22 return $k$-WST, $\phi, num_\phi, \text{start}_\phi$;

Figure 3.4: Expanding algorithm for $k$-word-based truncated suffix trees.
Procedure terminate;

Input: integers $i, k$, a text $T[1 \ldots i]$, a $k$-word-based suffix tree $k\text{-WST}(T[1 \ldots i - 1])$, a reference $\psi_i = (s, j, i - 1)$, the word count $num_\psi$ of $\psi$, and the start position $start_\psi$ of $\langle \psi_i \rangle$;

Output: A word based truncated suffix tree $k\text{-WST}(T)$, the reference $\psi_{i+1}$, the word count $num_\phi$ of $\psi_{i+1}$, and the start position $start_\phi$ of $\langle \psi_{i+1} \rangle$;

1 $\psi = \text{canonize}((s, j, i))$;
2 if $T[i] = \#$
3 $num_\psi := num_\psi + 1$;
4 while ($num_\psi = k$){
5 $t := \text{child}(s, T[j])$;
6 set the end position of $\text{label}(t)$ to $i$;
7 $\psi = \text{canonize}((\text{suf}(s), j, i))$;
8 if $T[i] = \#$
9 $num_\phi := num_\phi + 1$;
10 $start_\phi := start_\phi + 1$;
11 }
12 return $k\text{-WST}, \psi, num_\phi, start_\phi$;

Figure 3.5: terminating algorithm for $k$-word-based truncated suffix trees.
reference for the \((i+1)\)-th termination phase.

3.3 Analysis

Correctness

The following lemma indicates an important property of word-based truncated suffix trees.

**Lemma 1** Given any text \(T\) of length \(n\) and an integer \(k > 0\), let \(k\)-WST\((T)\) be the \(k\)-word-based truncated suffix tree for \(T\). Then, for any internal node \(u \in V\) in \(k\)-WST, the suffix link \(\text{suf}(u) \in V\) exists.

[Proof] Let \(a \in \Sigma\) be the character of \(\langle u \rangle\) and \(\alpha \subseteq \Sigma^*\) be the string such that \(\langle u \rangle = a \cdot \alpha\). Since \(u\) is an internal node, \(u\) has a child which represents a substring consists of at most \(k\) words. Then, for any child \(v \in V\) of \(u\), we have \(|a \cdot \alpha|_\# < ||_\# \leq k\) because \(|\langle u \rangle| < |\langle v \rangle|\).

On the other hand, let \(1 \leq i \leq n\) be the position of any occurrence of \(a \cdot \alpha\) in \(T\). Then, \(\alpha\) clearly occurs in \(T\) at the position \(i + 1\). Since \(|\alpha|_\# \leq |a \cdot \alpha|_\#\), there must exist the node which represents \(\alpha\).

Now, let \(s, t \in V\) be two children of \(u\). They represent strings \(a \cdot \alpha \cdot \beta (\beta \in \Sigma^+)\) and \(a \cdot \alpha \cdot \gamma (\gamma \in \Sigma^+)\), respectively. Hence there also exist two nodes \(x, y \in V\) which represent \(a \cdot \alpha \cdot \beta\) and \(a \cdot \alpha \cdot \gamma\), respectively. Then, the common parent \(z\) of \(x\) and \(y\), which represents the string \(\alpha\), must exists in \(V\). \(\square\)

Let \(\phi = (s, j, i)\) be a reference such that \(\langle \phi \rangle \in \text{Fac}(T)\) and \(|\langle \phi \rangle|_\# \leq k\). By Lemma 1, the suffix link to the node \(v = \text{suf}(s) \in V\) exists if \(s\) is an internal node. Then, the reference \(\phi' = (\text{suf}(s), j, i)\) refers a real or virtual node which represents the suffix of \(\langle \phi \rangle\) such that \(\langle \phi \rangle = a \cdot \langle \phi' \rangle\) for a character \(a \in \Sigma\). If \(s\) is a leaf, let \(u\) be the
parent of $s$ and $j'$ be an integer such that $\langle s \rangle = \langle u \rangle \cdot T[j' \ldots j - 1]$. Then, the reference \( \phi' = (\text{suf}(u), j', i) \) also refers a real or virtual node which represents the suffix of $\langle \phi \rangle$ such that $\langle \phi \rangle = a \cdot \langle \phi' \rangle$ for a character $a \in \Sigma$. If the suffix links for internal nodes are correctly constructed, \texttt{create} appends all the nodes like Ukkonen's algorithm and \texttt{terminate} closes all the leaves if they become strings of exactly $k$ words. Therefore the following theorem immediately holds.

**Theorem 3** Given an integer $k > 0$ and a text $T$ of length $n$, \texttt{construct-k-WST}, shown in Fig. 3.3, correctly constructs the word-based suffix tree $k$-WST($T$).

**Time complexity**

We first show the following lemma.

**Lemma 2** Given an integer $k > 0$ and a text $T$ of length $n$, the reference $\phi$ follows at most $n$ real nodes in the executions of the procedure \texttt{create} called by \texttt{construct-k-WST},

[Proof] Assume that $\phi = (s, j, i)$.

The reference $\phi$ follows a real node if it is not canonical. When the procedure \texttt{canonize} is called by \texttt{create} and we have the canonical reference $(t, j', i)$, $\phi$ follows at most $k = j' - j$ real nodes because any label has non-empty string.

Let $1 \leq i \leq n$ be any point in the construction. The \texttt{while} loop of \texttt{canonize} halts if $j > i$. We denote the value of $j$ after $i$-th execution of \texttt{canonize} by $j_i$ and also denote the number of nodes followed by $\phi$ by $\delta_i$. Then, we have

$$\delta_i \leq j_i - j_{i-1}$$
The summation of $\delta$ for all $1 \leq i \leq n$ is

$$\sum_{i=1}^{n} \delta_i = \sum_{i=1}^{n} (j_i - j_{i-1})$$

$$\leq (j_1 - j_0) + (j_2 - j_1) + \cdots (j_n - j_{n-1})$$

$$= j_n - j_0$$

Since $\phi$ is initialized by $(\text{root}, 1, 0)$, we have $j_0 = 1$. After $n$-th execution, we have $j_n \leq n + 1$ because canonize halts if $j > i = n$. Then, we have $j_n - j_0 = (n + 1) - 1 = n$.

As well as the proof of Lemma 2, we can show the following lemma.

**Lemma 3** Given an integer $k > 0$ and a text $T$ of length $n$, the reference $\psi$ follows at most $n$ real nodes in the executions of the procedure close called by construct-$k$-WST,

By Lemma 3 and 3, we show the following theorem.

**Theorem 4** Given an integer $k > 0$ and a text $T$ of length $n$, the procedure construct-$k$-WST correctly constructs $k$-WST($T$) in $O(n)$ time.

[Proof] We first consider the time complexity of the procedure create. Since $k$-WST($T$) has at most $n$ leaves, the while loop in line 2 is executed at most $n$ times through the construction of $k$-WST($T$). Lines 3 to 7 takes $O(1)$ time for each execution. Thus, except for the executions of canonize, Lines 2 to 9 takes $O(n)$ time in total. The while loop in line 15 is also executed at most $n$ times because the start position $\text{start}_\phi$ is incremented for each execution and $\text{start}_\phi \leq n + 1$. Therefore line 15 to 20 takes $O(n)$ time in total. Lines 1, 10, 11, 13, 14, 22 take $O(1)$, hence they take $O(n)$ time in total. By lemma 2, canonize takes $O(n)$ time in total. Then, the time complexity of create through the construction is $O(n)$ time.
Next we consider the time complexity of the procedure \texttt{terminate}. Lines 2 and 3 take $O(1)$ time. This procedure closes at most $n$ leaves in total. That is, the \texttt{while} loop is executed at most $n$ times in total. Except for the executions of \texttt{canonize} in line 7, lines 5 to 10 take $O(1)$ time. By lemma 3, \texttt{canonize} in line 1 and 7 takes $O(n)$ time in total. Overall, \texttt{terminate} takes $O(n)$ time through the construction.

Another components in \texttt{construct-k-WST} take $O(1)$ time. Therefore the time complexity of \texttt{construct-k-WST} is $O(n)$ time in total.

\section{3.4 Counting frequencies}

In applications of suffix trees, we often would like to need the frequency of each substring. In suffix trees, the frequency of the string represented by any leaf is exactly one because every leaf corresponds to a mutually different suffix. Then, the frequency of the substring represented by any internal node $u \in V$ is the number of leaves in the subtree rooted by $u$.

However, in word-based truncated suffix trees, leaf may correspond to more than one occurrence of the representing substring.

To compute correct frequency of any substring, we adopt a \texttt{counter} to each leaf. The value of each counter is initialized by one. In \texttt{create}, we increment the counter of a leaf $v$ if the reference $\phi$ reaches $v$. Then, the frequency of the string represented by a leaf is the value of its counter and the frequency of the substring represented by any leaf is the sum of the values of the counters of the leaves in the subtree rooted by $u$.

\section{3.5 Experimental results}

We compared the space consumption of the word-based truncated suffix trees with the normal suffix trees.
Data

 Reuters-21578\(^1\) is a test collection for text categorization. We removed the tag information from the original SGML[56] data. Then, the created test data \(T\) consisted of about 18.8M characters.

Method

We implemented the suffix tree and the word based suffix tree in C++. Constructing suffix trees and word based suffix trees for \(k = 1, 2, \ldots, 6\), we counted the numbers of nodes in them.

Results

Figure 3.6 shows the results. Compared with the normal suffix trees, word-based truncated suffix trees could drastically save the numbers of nodes with small \(k\)s. When \(k = 3\), the numbers of nodes were about a half of the normal one. On the other hand, word-based truncated suffix trees were almost the same as the normal one when \(k = 6\).

3.6 Discussion

In this chapter, we proposed the word-based truncated suffix tree and its efficient construction algorithm. Given an integer \(k > 0\) and a text \(T\) of length \(n\), we showed that the \(k\)-word-based suffix tree for \(T\) can be constructed in \(O(n)\) time, assuming that \(T\) is a string over a fixed size alphabet. We extended the algorithm to store the frequency on each leaf. Experimental results showed that the word-based truncated suffix tree could reduce the space consumption from the normal suffix tree.

\(^1\) http://www.daviddlewis.com/resources/testcollections/reuters21578/
Figure 3.6: The numbers of nodes in suffix trees and word-based truncated suffix trees.
It is a future work to extend the algorithm to a construction algorithm for word $N$-gram trees, which will appear in Chapter 5.
Chapter 4

Offline linear-time construction of property suffix trees

In this chapter, we study efficient construction of property suffix trees, which is an index structure proposed by Amir et al. [1]. Property is a set of intervals over a text. Given a text $T$ with a property $\pi$, the property suffix tree for $T$ and $\pi$ is an index structure that stores any substring in $T$ which is included by an interval in $\pi$. Amir et al. presented an $O(n \log|\Sigma| + n \log \log n)$ time construction algorithm. In this chapter, we focus attention on a subproblem that computing the borders between the outsides and the insides of the intervals. We then give an efficient algorithm for solving the subproblem and show that the property suffix tree can be constructed in $O(n \log |\Sigma|)$ time.

This chapter is based on the paper [50].

4.1 Text with property

Text with property $I = (T, \pi)$ is a pair of a string $T$ of length $n$ and a property $\pi$, where the property $\pi$ is a set $\pi = \{(s_1, f_1), \ldots, (s_m, f_m)\}$ such that $(s_i, f_i) \in \pi$ is
Figure 4.1: Text with property and queries.

$s_i, f_i \in \{1, \ldots, n\}$ かつ $s_i \leq f_i$ である。$1 \leq i \leq m$。$I$ の長さは同様に $n = |T|$ である。

We say that String $P$ is a substring of $T$ within $\pi$ if an interval $(s, f) \in \pi$ and a pair of positions $1 \leq i, j \leq n$ holds the following conditions: (i) $j = i + |P| - 1$ (ii) $P = T[i \ldots j]$ and (iii) $s \leq i$ および $j \leq f$。We also say that $P$ occurs at the position $i$ in $T$ under $\pi$。$Fac(T, \pi)$ denotes the set of substrings that occur under $\pi$，that is，$Fac(T, \pi) = \bigcup_{(s,f) \in \pi} Fac(T[s \ldots f]) \subseteq \Sigma^*$。

Example 1 Figure 4.1 shows a text with property $I = (T, \pi)$, where $T = ABABCBCABCBA$ and $\pi = \{(3, 4), (6, 9), (8, 12), (10, 13)\}$. As shown in the figure, position 9 is an occurrence of $P = ABC$ in $T$ under $\pi$ because $T[9 \ldots 11] = ABC$ is covered with the interval $(8, 12)$. On the other hand, position 3 is not because no interval in $\phi$ covers $(3, 9)$。

A property $\pi = \{(s_1, f_1), \ldots, (s_m, f_m)\}$ of $T$ is in standard form if it holds the following conditions:

1. For any $1 \leq i \leq n$，there is at most one $(s_k, f_k) \in \pi$ $(1 \leq k \leq m)$ such that $s_k = i$ and
2. $s_1 < s_2 < \cdots < s_m$。

That is, if $\pi$ is in standard form, all the intervals are given in an increasing order in
start positions and all the start positions are different. We notice that \( m \leq n \) always holds. For simplicity, we assume that any given property is in standard form.

### 4.2 Property suffix trees

This section introduces the property suffix trees, proposed by Amir et al.[1]. Let \( I = (T, \pi) \) be a text \( T \) of length \( n \) with a property \( \pi \). Then the property suffix tree for \( I \), denoted by \( P = PST(T, \pi) \), is a rooted tree \( P = (V', E', \text{root}, \text{child}, \text{suf}, \text{end}, \text{pos}) \) which is obtained by some modifications to the suffix tree \( S = ST(T) = \{V, E, \text{root}, \text{child}, \text{suf}\} \), including removals and merges of edges and additions of auxiliary information. \( V' \subseteq V \) is a set of real nodes. \( E' \subseteq V'^2 \) is a set of edges. The end-point function \( \text{end} \) returns a position for any \( 1 \leq i \leq n \). The position-set function \( \text{pos} \) returns a set of positions \( \text{pos}(s) \subseteq \{1, \ldots, n\} \) for each node \( s \in V' \).

**End-point function**

For each position \( 1 \leq i \leq n \) in \( T \), the end-point for \( i \), denoted by \( \text{end}(i) \) is defined as follows: \( \text{end}(i) \) is the maximal position \( f \) such that \( \sigma = (s, f) \in \pi \& \& s \leq i \leq f \). If no such \( \sigma \) exists, \( \text{end}(i) = i - 1 \). We define \( \text{end}(0) = -1 \).

The following lemma shows a nature of \( \text{end}(i) \) and is important for the correctness of the algorithm shown in Section 4.3:

**Lemma 4** Given a text with property \( I = (T, \pi) \) of length \( n \), \( \text{end}(0), \ldots, \text{end}(n) \) always forms a monotonically non-decreasing sequence. That is, \( \text{end}(0) \leq \cdots \leq \text{end}(n) \).

[Proof] Assume that \( \text{end}(i - 1) > \text{end}(i) \) for any position \( 1 \leq i \leq n \). Then, \( T_{\pi}^{i-1} = T[i - 1 \ldots \text{end}(i - 1)] \) covers \( T_{\pi}^{i} = T[i \ldots \text{end}(i)] \). By the definition of \( \text{end}(i) \), we have \( \text{end}(i) = \text{end}(i - 1) \). This leads to a contradiction. Thus, we have \( \text{end}(i - 1) \leq \text{end}(i) \) for any position \( 1 \leq i \leq n \). \( \square \)
$T = ABABCBCABCBAS$

$\pi = \{(3, 4), (6, 9), (8, 12), (10, 13)\}$

Figure 4.2: Relations between $\text{end}(\cdot)$ and $PSuf(T, \pi)$. 
The $i$-th suffix of $T$ with property $\pi$ is $T^i_\pi = T[i \ldots \text{end}(i)]$. $\text{PSuf}(T, \pi)$ denotes the set $\text{PSuf}(T, \pi) = \{T^i_\pi | 1 \leq i \leq n\}$ of the suffix of $T$ with property $\pi$.

**Example 2** Figure 4.2 shows an example of the arrays of end($\cdot$) and the set $\text{PSuf}(T, \pi)$ for a text with property $I = (T, \pi)$.

The following lemma is essential to the property matching problem.

**Lemma 5** Let $P$ be a pattern, $I = (T, \pi)$ be a text with property of length $n$, and $1 \leq i \leq n$ be any position in $T$. Then, $P$ occurs in $T$ at the position $i$ under $\pi$ iff $P$ is a prefix of $T^i_\pi$.

Lemma 5 indicates that the data structure that indexes $\text{PSuf}(T, \pi)$ is the key to solve the property matching problem.

**Lower set queue**

Let $(A, \leq)$ be any totally ordered set, where $A = \{a_1, \ldots, a_n\}$ is a set of elements and $\leq$ is a total order on $A$. We denote the size of $A$ by $|A|$. For any elements $x, y \in A$, we say that $y$ is lower than $x$ if $x \leq y$.

**Definition 1 (Lower set queue)** Let $(A, \leq)$ be any totally ordered set. A lower set queue of $(A, \leq)$ is an abstract data structure that stores $S \subseteq A$:

- **create-LQ$(S)$**: Given a set $S$, it returns the lower set queue $\text{LQ}(S)$ of $S$.

- **lower$(LQ(S), x)$**: Given a lower set queue $\text{LQ}(S)$ of a set $S$ and an element $x \in A$, it returns the lower set $\text{Lower}(S, x) = \{y \in S \mid x \leq y\}$.

**Theorem 5 (Amir et al.[1])** Let $(A, \leq)$ be a totally ordered where the comparison of $\leq$ can be done in $O(1)$. Then, for any set $S \subseteq A$ and any element $x \in A$, we can implement the lower set queue $\text{LQ}(S)$ of $S$ which computes $\text{LQ}(S) = \text{create-LQ}(S)$ in $O(|S|)$ time and also computes $\text{lower}(\text{LQ}(S), x)$ in $O(|\text{Lower}(S, x)|)$ time.
Figure 4.3 and 4.4 shows the procedures **create-LQ** and **lower** that suffice Theorem 5, respectively. The sub-procedure **split-set**\((S)\) divides \(S\) into two sets \(LH = \{x \in S | x \leq M_S\}\) and \(UH = \{x \in S | M_S \leq x\}\), where \(M_S\) is the median of \(S\). The time complexity of **split-set**\((S)\) is \(O(|S|)\) time. We call a lower set queue is **efficient** if it suffices Theorem 5.

**Position set function**

Let \(S = ST(T)\) be the suffix tree for an input text \(T\) and \(Pos = \{1, \ldots, n\}\) be a set of positions. In the suffix tree \(S\), each suffix \(T^i = T[i \ldots n]\) of \(T\) is represented by a leaf. On the other hand, in the property suffix tree \(P\) for \(I = (T, \pi)\), each property suffix \(T^i_\pi = T[i \ldots end(i)]\) is represented by a real or virtual node \(x\). However, the virtual node are not explicitly represented. Therefore we will describe an idea to store the positions corresponding to virtual nodes. We identify a substring \(w\) of \(T\) as the corresponding node \(w\) obtained by traversing \(S\) with the characters of \(w\).

First, for each real or virtual node \(x \in Fac(T, \pi)\), we associate \(x\) with the set
\[
pos_1(x) = \{ i | T^i_\pi = x \}.
\]
This set consists of the positions of the property suffixes represented by \(x\). Second, for each real node \(s \in V\), we define a set \(X(s) = Pre(\overline{t}) \setminus Pre(t)\), where the real node \(t \in V\) is the parent of \(s\). By the definition of set \(X\), the elements in \(X(s)\) corresponds to the all strings belong to \(s\). Finally, for any real node \(s \in V^r\), we define \(pos(s) = \bigcup_{x \in X(s)} pos_1(x) \subseteq Pos\). The **depth** of a position \(i \in pos(s)\), denoted by \(dep(i)\), is defined as \(dep(i) = |T^i_\pi|\).

We define the total order \(\leq_{dep}\) as follows: for two positions \(i, i' \in pos(s)\), \(i \leq_{dep} i'\) iff \(dep(i) \leq dep(i')\). The set \(pos(s)\) is represented by an efficient lower set queue for \((Pos, \leq_{dep})\).
Procedure create-$LQ$;

Input: A set $S = \{a_1, \ldots, a_n\}$ where $n \geq 0$;

Output: The lower set queue $LQ(S)$;

1. $LQ = LQ(\emptyset)$;
2. $S' = S$;
3. while $S' \neq \emptyset$
   4. $(LH, UH) = \text{split-set}(S')$;
   5. foreach $a \in UH$
      6. Append $a$ to the head of $LQ$;
   7. $S' = LH$;
4. };
9. return $LQ$;

Figure 4.3: A procedure for constructing lower set queues.
Procedure lower;

Input: A lower set queue \( LQ(S) \), \( x \in A \);

Output: The lower set \( L = Lower(LQ(S), x) \);

1. \( L := \emptyset \);
2. \( i := 1 \);
3. \( flag := true \);
4. \( \text{while } flag = true \{ \)
   
   5. \( \text{for } j = 1, \ldots, 2^{i-1} \{ \)
      
      6. Take the first element \( a \) of \( LQ(S) \);
      
      7. \( \text{if } x \leq a \)
      
      8. \( L := L \cup \{a\} \);
      
      9. \( \text{else} \)
      
      10. \( flag := false \);
      
   
   11. \} \)
   
   12. \( i := i + 1 \);
   
   13. \} \)
5. \( \text{return } L \);

Figure 4.4: A procedure for computing lower sets.
Figure 4.5: The property suffix tree for the text with property shown in Fig. 1.
Real nodes and edges

Now we define the set $V'$ of the real nodes and the set $E'$ of the edges of property suffix trees. Given a suffix tree $S$, we assume that the computation of pos is done. A real node $v$ is a *border* if $\text{pos}(u)$ is nonempty. Section 4.3 will give an equivalent definition of borders. Then, $V'$ and $E'$ can be obtained as follows: First we mark all the borders and their ancestors. Second, we eliminate all the real nodes which are not marked and all the edges connecting to them. Then, we eliminate the nodes which have only one child and merge and the edges connecting to the nodes. We also merge the position sets pos correspond to the edges at this time. The procedures *prune* and *adjust* introduced in Section 4.3 describes the details of this process. Finally, for each leaf $v \in V'$, $\text{label}(v)$ is modified to represent $\langle v \rangle = T^i_n$ if $|\langle v \rangle| > |T^i_n|$, where $i$ is the position which have the maximal dep in pos($v$). Then we have the compete sets $V'$ and $E'$ and also have the property suffix tree $P = \text{PST}(T, \pi)$ for $I = (T, \pi)$.

Figure 4.5 shows the property suffix tree $\text{PST}(T, \pi)$ for the text with property $I = (Tm\pi)$ shown in Fig. 4.1.

Property matching problem

Let $I = (T, \pi)$ be a text with property of length $n$ and $P$ be a query of length $m$. The *property matching problem* is a problem to compute all the occurrence $i_1, \ldots, i_{\text{occ}} \subseteq \{1, \ldots, n\}$ of $P$ in $T$ under $\pi$.

This problem can be efficiently solved by some pattern matching algorithms, such as KMP algorithm [29] and Boyer-Moore algorithm [5]. However, it is non-trivial that how to construct an index structure which efficiently solves them repeatedly.

Given the property suffix tree $P = \text{PST}(T, \pi)$ and a pattern $P$ of length $m$, the property matching problem can be solved as follows: First we traverse $P$ with $P$. If traversal fails, there is no occurrence. Otherwise, let $u \in V'$ be the real node such that
$P \in X(u)$. Next, by a depth-first traversal, we output all the elements in $pos(v)$ for any node in the subtree rooted by $u$. Finally, using the lowser set queue $pos(u)$, we output each element $i \in pos(u)$ such that $dep(i) \geq m$, that is, $Lower(pos(s), m)$. Using the property suffix tree, we can also solve the weighted matching problem, stated in [1].

Amir et al. [1] showed the following lemma about the property matching problem:

**Lemma 6** Let $I = (T, \pi)$ be a text with property. Given the property suffix tree $PST(T, \pi)$ and a pattern $P$ of length $m$, the property matching problem can be solved in $O(m \log |\Sigma| + tocc)$ time, where $|\Sigma|$ is the size of the alphabet and tocc is the number of occurrences of $P$ in $T$ under $\pi$.

### 4.3 Proposed algorithm

**Outline of the construction algorithm**

Figure 4.6 shows the procedure `construct-PST` that constructs the property suffix tree $PST(T, \pi)$ for a given text with property $I = (T, \pi)$. This procedure first constructs the suffix tree $ST(T)$. Second, it computes the end-point $end(i)$ for each $1 \leq i \leq n$. Third, it determines the nodes which are the borders between the elements in $V'$ and the other nodes. Then it eliminates the nodes $V \setminus V'$ from $ST(T)$. Finally it modifies the labels of the edges connecting to the leaves.

**Computing borders**

Let $I = (T, \pi)$ be a text with property of length $n$. The *border* for a position $1 \leq i \leq n$, denoted by $bd(i)$, is the real node $s \in V$ such that $T_i^s \in X(s)$. We notice that there is exactly one $bd(i)$ in the path from the *root* to the leaf $t$ which represents the suffix $T[i \ldots n]$. Figure 4.7 shows the procedure `property-tour` which computes $bd(i)$ for
Procedure construct-$PST$;

**Input:** a text with property $I = (T, \pi)$ of length $n$;

**Output:** The property suffix tree $PST(T, \pi)$ for $I$;

1. $PST = ST(T)$;
2. for $i = 1$ to $n$
3. \hspace{1em} end($i$) = max($\{f | (s, f) \in \pi, s \leq i \leq f\}$ $\cup \{i - 1\}$);
4. property-tour($PST$, end($\cdot$));
5. prune-PST($PST$);
6. adjust($PST$);
7. return $PST$;

Figure 4.6: A construction algorithm for property suffix trees.
all positions \(i = 1, \ldots, n\) from a suffix tree \(ST(T)\) and a property \(\pi\). This procedure stores the reference \(w\) which corresponds to \(T_\pi^i\) and computes \(bd(i + 1)\) by using the reference. The set \(pos(s)\) for each real node \(s \in V'\) is implemented by a linked list.

By lemma 4, the difference between \(T_\pi^{i-1}\) and \(T_\pi^i\) are the first character \(T[end(i-1)+1 \ldots end(i)]\). Suppose that \(\langle bd(i-1) \rangle = T_\pi^{i-1}\). Then, since the suffix link \(suf(bd(i-1))\) from \(bd(i-1)\) to \(T[i \ldots end(i-1)]\) always exists, \(bd(i)\) can be found by following with the difference string \(T[end(i-1)+1 \ldots end(i)]\) from \(suf(bd(i-1))\). If \(T_\pi^{i-1}\) is represented by a virtual node \(v\), we can also find \(bd(i)\) by using the nearest real ancestor \(s\) of \(v\) and its suffix link \(suf(s)\).

Figure 4.9 shows an example of the move of the reference \(\phi\) in the procedure \texttt{property-tour}. The moves to lower positons correspond to the transitions by suffix links and moves to rightward positions correspond to the transitions by following with the difference strings. When the reference \(\phi = (s,j,i)\) moves to the reference \(\phi = (s',j',i')\) which corresponds to its suffix, the procedure \texttt{canonize} follows only real nodes.

**Eliminating nodes from suffix trees**

The procedure \texttt{prune-PST} shown in Fig. 4.10 eliminates the nodes \(V \setminus V'\) from a suffix tree \(ST(T)\). Let \(u\) be any node \(u \in V \setminus V'\) which has only one child \(v\). To keep the tree to be path-compacted, we merge the two edges connecting \(u\) into one edge and also merge \(pos(u)\) into \(pos(v)\). Then, we construct the lower set queue for each node in \(V'\).

Finally, applying the procedure \texttt{adjust} shown in Fig. 4.11, the label of any leaf \(v \in V'\) is modified such that \(\langle v \rangle \in PSuf(T,\pi)\). That is, for each leaf \(v \in V'\), we cut some suffix of \(label(v)\) such that

\[|\langle v \rangle| = \max\{end(i) - i \mid i \in pos(s)\}.\]
Procedure property-tour;

Input: A suffix tree $ST(T)$ and a set of end-points $\{\text{end}(1), \ldots, \text{end}(n)\}$;

1 $\phi = (s, j, i) = (\text{root}, 1, 0)$;
2 for $i = 1$ to $|T|$ {
3     $\phi = \text{canonize}((s, j, \text{end}(i)))$
4     if $j > \text{end}(i)$
5         $\text{bd}(i) = s$
6     else
7         $\text{bd}(i) = \text{child}(s, T[j])$
8     $\text{pos}(\text{bd}(i)) = \text{pos}(\text{bd}(i)) \cup \{i\}$
9     $t = \text{bd}(i)$
10    while $t \neq \text{root}$ and $t$ is not marked{
11        Mark $t$
12        $t = t$’s parent;
13    }
14 }

Figure 4.7: A procedure for computing the borders.
4.4 Analysis

Correctness

We first show that the procedure \texttt{property-tour} computes the borders $bd(1), \ldots, bd(n)$ if $end(1), \ldots, end(n)$. The following lemma is essential to the correctness of the procedure.

Lemma 7 Let $I = (T, \pi)$ be a text with property of length $n$, $ST(T)$ be the suffix tree for $T$, and $\phi_i = (s, j, i)$ be a reference corresponding to $bd(i)$ for $1 \leq i \leq n$. Then, for any $1 < i \leq n$, $\phi_i = (suf(s), j, i) = bd(i)$ if $\phi_i = (s, j, i - 1) = bd(i - 1)$.

[Proof] By the definition of borders, we have $\langle bd(i - 1) \rangle = T^{i-1}_n$ and $\langle bd(i) \rangle = T^i_n$ for any position $1 < i \leq n$. Since $\langle bd(i - 1) \rangle = \langle s \rangle \cdot T[j \ldots end(i - 1)]$, we have $\langle s \rangle = T[i - 1 \ldots j - 1]$. By the definition of suffix links, we have $\langle suf(u) \rangle = T[i \ldots j - 1]$. Then,
we have \( \langle \text{suf}(u), j, \text{end}(i) \rangle = T[i \ldots j - 1] \cdot T[j \ldots \text{end}(i)] = T[i \ldots \text{end}(i)] = \langle \text{bd}(i) \rangle \).

Thus, for any position \( 1 < i \leq n \), we have \( \langle \text{suf}(s), j, \text{end}(i) \rangle = \text{bd}(i) \).

Lemma 7 immediately leads the following corollary by induction.

**Lemma 8** Given the suffix tree \( \text{ST}(T) \) for a text \( T \) and a list of end-points \( \text{end}(1), \ldots, \text{end}(n) \), the procedure property-tour correctly computes all the borders \( \text{bd}(1), \ldots, \text{bd}(n) \).

Now we proof the correctness of the construction algorithm for property suffix trees.

**Theorem 6** Given a text with property \( I = (T, \pi) \) of length \( n \), the procedure construct-PST correctly constructs the property suffix tree \( \text{PST}(T, \pi) \).

**[Proof]** Lemma 8 ensures the correctness of the procedure property-tour. After the computation of the borders, the procedures prune-PST and adjust clearly constructs the correct property suffix tree defined in Section 4.2.

**Time complexity**

We first show the time complexity of the procedure property-tour. For a node \( v \) in a suffix tree \( \text{ST}(T) \), \( \text{nodepath}(v) \) denotes the path of real nodes \( \text{root}, u_1, \ldots, u_k = v \) from \( \text{root} \) to \( v \) if \( v \) is a real node. If \( v \) is a virtual node, \( \text{nodepath}(v) = \text{root}, u_1, \ldots u_k = p \), where \( p \) is the parent of \( v \). For a node \( v \) in \( \text{ST}(T) \), \( \text{nodedepth}(v) \) denotes the number of
Procedure prune-PST;
Input: A suffix tree $ST(T)$ whose borders and their ancestors are marked;
1    foreach $\{u \in V'|u$ is not marked$\}$ do
2        Delete the subtree rooted by $u$ from $ST(T)$;
3        if $u$’s parent $t$ has just one child $v$ then
4            $pos(v) = pos(v) \cup pos(t);$  
5            Delete $t$ and merge the connecting edges to $t$;
6        end if
7    end do
8    for each $u \in V'$
9        Construct the lower set queue $pos(u)$;

Figure 4.10: A procedure for pruning nodes.

Procedure adjust;
Input: A tree $PST^\pi(T, \pi)$ created by construct-PST;
1    for each $\{v \in V'|v$ is a leaf in $PST(T, \pi)\}$ do
2        $i = \max(\{dep(i) \mid i \in pos(v)\});$
3        if $|\langle v \rangle| > dep(i)$
4            Modify label($v$) such that $\langle v \rangle = T^i_{\pi};$
5    }
real nodes in $\text{nodepath}(v)$. We define that $\text{nodedepth}(\text{root}) = 0$. We show the following lemma.

**Lemma 9** Let $T$ be a text, $ST(T)$ be the suffix tree for $T$, and $v$ be a real node in $ST(T)$. Then, $\text{nodedepth}(\text{suf}(v)) \geq \text{nodedepth}(v) - 1$.

**Proof** Let $m$ be the number of real nodes in $\text{nodepath}(v)$. By the definition of suffix trees, the real node $\text{suf}(v)$ always exists for any node in the path $\text{nodepath}(v)=\text{root}, u_1, \ldots, u_{m-1}$. Then, $\text{nodepath}(\text{suf}(v))$ includes the nodes $\text{suf}(u_1), \text{suf}(u_2), \ldots, \text{suf}(u_{m-1})$, except for $\text{suf}(\text{root}) = \perp$. Thus we have $\text{nodepath}(\text{suf}(v)) \geq m - 1 = \text{nodepath}(v) - 1$.

In a similar way to Lemma 9, we show the following lemma for virtual nodes.

**Lemma 10** Let $T$ be a text, $ST(T)$ be the suffix tree for $T$, $w$ be a real or a virtual node in $ST(T)$, and $x$ be the node which represents the suffix of length $|\langle w \rangle| - 1$. Then, $\text{nodedepth}(x) \geq \text{nodedepth}(w) - 1$.

We denote $\text{nodedepth}(\text{bd}(i))$ by $d(i)$. For the $i$-th iteration of the property tour, $\Delta(i)$ denotes the number of the real nodes followed by the execution of canonize. Then, we show the following lemma.

**Lemma 11** Let $I = (T, \pi)$ be a text with property of length $n$ and $ST(T)$ be the suffix tree for $T$. Then, $\sum_{i=1}^{n} \Delta(i) \leq n + 1$.

**Proof** We have $|\langle \text{bd}(i) \rangle| \leq n$ because $0 \leq \text{end}(i) \leq n$. Since any label in the suffix tree is nonempty, we have $d(i) \leq |\langle \text{bd}(i) \rangle| \leq d(i) \leq n$. By Lemma 10, we have $\text{nodedepth}(v) \geq d(i - 1) - 1$ where $v$ is the node that represents the suffix of $\langle \text{bd}(i) \rangle$ with length $|\langle \text{bd}(i) \rangle| - 1$. By the definition of $\Delta$, we have $d(i) = \text{nodedepth}(v) + \Delta(i)$. Then, we have $d(i) = \text{nodedepth}(v) + \Delta(i) \geq d(i - 1) - 1 + \Delta(i)$. Since $\text{end}(0) = -1$,
we have $T[0 \ldots -1] = \varepsilon$ and then $d(0) = 1$. Since $\text{end}(n) \leq n$, we have $d(n) \leq 2$. The summation of $d(i)$ for $i = 1, \ldots, n$ is:

$$\sum_{i=1}^{n} d(i) \geq \sum_{i=1}^{n} \{ d(i) - 1 + \Delta(i) \}$$

$$= \sum_{i=1}^{n} d(i - 1) - n + \sum_{i=1}^{n} \Delta(i),$$

$$\therefore \sum_{i=1}^{n} \Delta(i) \leq \sum_{i=1}^{n} d(i) - \sum_{i=1}^{n} d(i - 1) + n$$

$$= d(n) - d(0) + n \leq n + 1.$$  

Therefore we have $\sum_{i=1}^{n} \Delta(i) \leq n + 1$. \hfill \Box

Using Lemma 11, we show the following theorem.

**Lemma 12** Let $I = (T, \pi)$ be a text with property of length $n$. Then, the time complexity of the procedure property-tour shown in Fig. 4.7 is $O(n \log |\Sigma|)$ time.

[Proof] Line 1 can be done in $O(1)$ time. The time complexity of the procedure canonize is proportional to the number of iterations in the while loop in Fig. 2.3. By Lemma 11, canonize follows at most $n + 1$ real nodes in total. Since a child can be found in $O(\log |\Sigma|)$ time by using red-black trees[12], the total computation time of canonize called by line 3 in Fig. 4.7 is $O(n \log |\Sigma|)$ time. Line 4 and 5 can be done in $O(1)$ time. Line 7 can be done in $O(n \log |\Sigma|)$ time by finding a child from red-black trees. Thus, the time complexity of lines 4 to 7 is $O(n \log |\Sigma|)$ time.

Next we consider about lines 10 to 13. Since the while loop in line 10 halts if an ancestor is marked, any real node is marked at most once. Therefore the total number of marked nodes is at most $2n - 1$ and the time complexity of lines 10 to 13 is $O(n)$. Then, the time complexity of property-tour is $O(n \log |\Sigma|)$ time. \hfill \Box

Now we show the main result of this chapter.

**Theorem 7** Let $I = (T, \pi)$ be a text with property of length $n$. Then, the procedure construct-PST constructs the property suffix tree $PST(T_{\pi})$ in $O(n \log |\Sigma|)$ time.
[Proof] The construction of the suffix tree in line 1 takes $O(n \log |\Sigma|)$ [54].

Recall that we assume $\pi$ is in standard form. For each position $1 \leq i \leq n$, the position $\text{end}(i)$ is obtained by comparing $\text{end}(i-1)$, $i-1$ and $f_i$ of an interval $(i, f_i)$ if such $(i, f_i)$ exists. Therefore the time complexity of lines 2 and 3 is $O(n)$ time. By Lemma 12, line 4 takes $O(n \log |\Sigma|)$ time.

Next we consider the procedure $\text{prune-PST}$ shown in Fig. 4.10. In line 2 in $\text{prune-PST}$, let $\text{del}(u)$ be the number of the nodes in the deleting subtree rooted by a node $u \in V$ which is not marked. Then, deleting the subtree takes $O(\text{del}(u))$ time. Line 3 takes $O(1)$ time. In line 4, since there is exactly one border $v \in V'$ for any position $1 \leq i \leq n$, we have $\text{pos}(u) \cap \text{pos}(v) = \emptyset$ for any real nodes $u, v \in V'$. Thus, the operation $\text{pos}(u) \cup \text{pos}(v)$ is just a concatenation of two linked lists, and takes $O(1)$ time. Line 5 takes $O(1)$ time. Therefore the time complexity of lines 2 to 6 is $O(\text{del}(u))$ time. Let $k$ be the number of iteration of the $\text{foreach}$ loop in line 1. Lines 1 to 7 take $\sum_{i=1}^{k} (\text{del}(s) + 1) = \sum_{i=1}^{k} \text{del}(s) + k$. For any position $1 \leq i \leq k$, the node $s_i = u$ has not been deleted in the previous iterations. Since $\text{ST}(T)$ has at most $2n - 1$ nodes, we have $\sum_{i=1}^{k} \text{del}(s) \leq 2n$. Then, the time complexity of lines 1 to 7 is $O(n)$ time. By Theorem 5, for each real node $u \in V'$, the construction of the lower set queue $\text{pos}(u)$ takes $O(|\text{pos}(u)|)$ time. Therefore the procedure $\text{prune-PST}$ takes $O(n)$ time.

Finally, we consider the time complexity for the procedure $\text{adjust}$ shown in Fig. 4.11. For any leaf $v \in V'$, the computation of the position which has the maximal depth $\text{nodedepth}(i)$ from $i \in \text{pos}(v)$ takes $O(|\text{pos}(v)|)$ time. Since the total number of positions is at most $n$, the executions of line 2 take $O(\sum_{v \in V'} |\text{pos}(s)|) = O(n)$ time in total. Line 3 and 4 takes $O(1)$ time. Thus, the time complexity of $\text{adjust}$ is $O(n)$ time.

Therefore, we conclude that $\text{construct-PST}$ runs in $O(n \log |\Sigma|)$ time in total. \qed
4.5 Experimental results

We implemented the proposed algorithm and compared with two naive algorithms by using artificial data.

Data

We created texts on the alphabet $\Sigma = \{A, G, C, T\}$. We created different properties for each experiment.

Method

We measured the computation time for computing the borders. The algorithms are three:

- **suf**: the proposed method,
- **root**: a naive method which searches the border by following labels from the root, and
- **leaf**: a naive method which searches the border $bd(i)$ from the leaf corresponds to $T[i \ldots n]$.

Experiment 1

The text were created by a uniform random distribution. The property $\pi$ is $\{(i, i + 10)|1 \leq i \leq n - 10\}$.

Experiment 2

The text were created by the probability distribution $Pr(A) = Pr(T) = 0.4$ and $Pr(G) = Pr(C) = 0.1$. The property $\pi$ is the same as Experiment 1.
Experiment 3

The text were $T = \text{AAA} \cdots \text{A}$. The property $\pi$ is $\{(i, \min(i + \lfloor (n - i)/2 \rfloor, n))|1 \leq i \leq n\}$. That is, the length of each property suffix $T_i^\pi$ is a half of the length of the suffix $T[i \ldots n]$.

Results

Result 1

Figure 4.12 shows the results of Experiment 1. The algorithm $\text{leaf}$ was the fastest when the input text is small. However, $\text{suf}$ was the best when $n > 1000000$.

Result 2

Figure 4.13 shows the results of Experiment 2. Compared with the results in Experiment 1, the results of $\text{root}$ and $\text{suf}$ were almost same. On the other hand, $\text{leaf}$ became slow.

Result 3

Figure 4.14 shows the results of Experiment 3. The result of $\text{suf}$ is almost same as Experiment 1. However, the computation time of $\text{root}$ and $\text{leaf}$ seems to be quadratic, which is the worst case time complexity of the algorithms.

4.6 Discussion

In this chapter, we studied efficient construction of property suffix trees. The construction algorithm presented by Amir et al. [1] takes $O(n \log |\Sigma| + n \log \log n)$ time for the border nodes. To improve the process, we gave an efficient algorithm for finding the
border nodes and showed that the property suffix tree can be constructed in $O(n \log |\Sigma|)$ time. However, an online construction algorithm is still open.
Figure 4.13: Results for the text created by the probability distribution \( \{P(A) = P(T) = 0.4, P(C) = P(T)0.1\} \).
Figure 4.14: Results for the text $T = \text{AAA} \cdots \text{A}$.
Chapter 5

Keyword extraction and browsing support

In this chapter, we propose the word $N$-gram tree, which is an extension of the word-based truncated suffix tree. The word-based truncated suffix tree, proposed in Chapter 3, indexes not only all the sequences of at most $k$ words but also their substrings. It is not a desirable for some applications. On the other hand, there is another word-based index structure, called the word suffix tree, proposed by Anderson et al. [3] and showed its online construction algorithm by Inenaga et al. [21]. Given a text $T$, the word suffix tree stores any substring which starts at a head of a word in $T$. In other words, the constraint considered in the word suffix tree limits the start positions of the substrings, but do not limit the end positions. The word $N$-gram tree is therefore an extension of both the word-based truncated suffix tree and the word suffix tree. In this chapter, we also consider a keyword extraction problem for Web browsing using the word $N$-gram trees. In information gathering from Web pages, there are various candidates for keywords, such as titles of books or movies, slang terms, and newly-coined expressions. We propose a keyword extraction algorithm using word $N$-gram trees and discuss the keyword extraction from a book and blogs.
This chapter is based on the paper [49].

5.1 Keyword Extraction Problem

Input Data

We first describe the input data for our keyword extraction algorithm. Let $T_0$ be the text part of a Web page. We assume that each word in $T$ ends at a delimiter #. Generally, Web pages contain some unusual words, such as “next” or “read more”. We therefore would like to remove them from $T_0$. Let $P$ be any paragraph in $T_0$. In HTML pages, we regard each block-level element as a paragraph. Then, given a threshold value $\delta_0 (0 < \delta_0 < 1)$, we remove $P$ from $T_0$ if the percentage of punctuations in $P$ is less than $\delta_0$. The input text $T$ for our keyword extraction algorithm is then the concatenation of all the remaining paragraphs.

Measuring Keywords

Given an input text $T = w_1 \cdots w_m$, a candidate for a keyword is an nonempty word sequence $X = w_i \cdots w_j (1 \leq i \leq j \leq m)$ which occurs in $T$. The rest of this subsection describes the measure for candidates.

First, we define the part-of-speech score for any part-of-speech. We assume that any part-of-speech score is greater than or equel to zero. In keyword extraction, nouns are relatively more important than the others in general. Table 5.1 shows the list of part-of-speech scores that is to be used in the later experiments. We take the most specific one if a word matches more than one part-of-speech. For example, given words “string” and “that”, we have $C(“string”) = 10$ and $C(“that”) = 0$.

Second, we define the score for each word. Let $x$ be any word of length $m$. Then,
Table 5.1: An example of the part-of-speech scores.

<table>
<thead>
<tr>
<th>part-of-speech</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>noun</td>
<td>10</td>
</tr>
<tr>
<td>noun (suffix)</td>
<td>1</td>
</tr>
<tr>
<td>noun (number)</td>
<td>1</td>
</tr>
<tr>
<td>noun (bound)</td>
<td>1</td>
</tr>
<tr>
<td>noun (pronoun)</td>
<td>0</td>
</tr>
<tr>
<td>verb</td>
<td>2</td>
</tr>
<tr>
<td>verb (suffix)</td>
<td>1</td>
</tr>
<tr>
<td>verb (bound)</td>
<td>1</td>
</tr>
<tr>
<td>adjective</td>
<td>2</td>
</tr>
<tr>
<td>adjective (suffix)</td>
<td>1</td>
</tr>
<tr>
<td>adjective (bound)</td>
<td>1</td>
</tr>
<tr>
<td>auxiliary</td>
<td>1</td>
</tr>
<tr>
<td>particle</td>
<td>1</td>
</tr>
<tr>
<td>adverb</td>
<td>1</td>
</tr>
<tr>
<td>prefix</td>
<td>1</td>
</tr>
<tr>
<td>other</td>
<td>0</td>
</tr>
</tbody>
</table>
the word score for $x$ is defined as

$$W(x) = C(x) \cdot (|x| - 1).$$

In oriental languages, there are some kinds of characters, such as Kanji, Hiragana, and Katakana in Japanese. These characters sometimes have more information than another characters. Therefore we adjust the word scores for such characters as the following example.

**Example 3** For Kanji, Hiragana, Katakana and the other characters in Japanese texts, we regard the length of each character as 4, 2, 2, 1, respectively.

By using word scores, we define the score for any word sequence. The phrase score for any word sequence $X = x_1 \cdots x_k$ in $T$, denoted by $P(X)$, is defined as

$$P(X) = \sum_{i=1}^{k} W(x_i).$$

Part-of-speech score, word score, phrase score are defined by each string itself. In other words, they are independent from any input text. Figure 5.1 shows an example of the computation of a phrase score.

Finally, we now define the score for any candidate in the input text. For any candidate $X$ in $T$, the keyword score $PF(X)$ is defined as

$$PF(X) = P(X) \cdot \log(f(T, X)).$$

In the rest of this chapter, we call the keyword score for $X$ just the score.

Since any part-of-speech score is non-negative, we have the following lemma.

**Lemma 13** Let $X, Y$ be sequences of words and $Y$ is a prefix of $X$. Then, $P(X) \geq P(X')$ holds.

By Lemma 13, we can observe that $PF(X) \geq PF(Y)$ if $f(T, X) = f(T, Y)$ for any candidate $X$ and its prefix $Y$ in $T$. We say that $Y$ is typified by $X$. Then, we define the
candidate set $K(T, N)$ as the set of any candidate $X$ of at most $N$ words such that there is no candidate $Y$ such that $X$ is a prefix of $Y$, $|X|_\# > |X'|_\#$ and $f(T, X) = f(T, X')$.

We then state our keyword extraction problem as follows.

Word $N$-gram keyword extraction problem: Given an input text $T$ and a score for candidates, output top-$k$ scored candidates in $K(T, N)$.

5.2 Truncated Word Suffix Trees

This section introduces the word $N$-gram tree which is the index structure to be used in our keyword extraction algorithm.

For simplicity, we assume a fixed size alphabet $\Sigma$ in this chapter. Given a text $T = w_1 \cdots w_m (w_i \in \Sigma^+ \cdot \# \text{ for } 1 \leq i \leq m)$ and an integer $N > 0$, the word $N$-gram set for $T$ is the set $WFac(T, N) = \{w_i \cdots w_j \mid 1 \leq i \leq m, j \leq \min(m, i + N - 1)\}$.

The main difference between word $N$-gram tree and word-based truncated suffix tree is that word $N$-gram tree only represents all the word sequences in $T$, while word-based truncated suffix tree represents all the word sequences and their substrings. That is, the word $N$-gram tree for $T$, denoted by $N$-TWST($T$), is the path-compacted trie that represents $WFac(T, N)$.

More formally, $N$-TWST($T$) = $\{V, root, E, freq\}$ is the rooted tree that satisfies the following conditions:

- $V$ is a set of nodes.
- $root \in V$ is the root of $N$-TWST($T$).
- $E$ is a set of edges.
- $\bot$ is a special node $\bot \notin V$ which is the parent of $root$.

Any internal node has at least two children. All the out-going edges leaving from a vertex always have labels which start with mutually different characters. We denote the
corresponding label of the ingoing edge of a node \( v \) by \( \text{label}_v \). Any label are represented by a pair of positions \((i,j)\), where \( \text{label}_v = T[i..j](1 \leq i, j \leq |T|) \) for each node \( v \in V \).

Each vertex \( v \in V \) represents a unique substring \( \langle v \rangle \in WFac(T, N) \), where \( \langle v \rangle \) is the longest prefix of the concatenation of the labels on the unique path from the root to \( v \), which ends at \( \# \). If the concatenation of the labels does not include \( \# \), we define \( \langle v \rangle = \varepsilon \). Any leaf represents a word sequence of \( N \) words or a suffix of \( T \). We denote the node which represents a string \( P \) by \( P \). If such node does not exist, \( P \) is undefined.

For any node \( v \in V \), the function \( \text{freq}(v) \) returns the frequency of \( \langle v \rangle \) in \( T \). Since \( v \) represents a sequence of words in \( T \), including \( \varepsilon \), \( \text{freq}(v) > 0 \) always holds for any node \( v \in V \).

Let \( u, v \) be nodes in \( V \) and \( x \) is any prefix of \( \langle v \rangle \). Then, we say that \( x \) belongs the edge \((u,v)\) iff \( |\pi|_\# < |x|_\# < |v|_\# \).

Since \( N\text{-TWST}(T) \) has at most \( m \) different start positions of words, the number of leaves is at most \( m \) and the number of nodes in \( V \) is at most \( 2m - 1 \). Each node takes \( O(\log n) \) space for storing two positions in \( T \). Then, we have the following theorem.

**Theorem 8** Let \( T = w_1 \cdots w_m \) be a text of length \( n \) and \( N > 0 \) be an integer. Then, the word \( N \)-gram tree for \( T \) can be constructed in \( O(m \log n) \) space.

About the frequencies of the subphrases in the text, we have the following lemmas.

**Lemma 14** Let \( N > 0 \) be an integer, \( T \) be a text, and \( \text{TWST}(T,N) \) be the word \( N \)-gram tree for \( T \). Then, for any node \( v \in V \), \( v \)'s parent \( u \), and any phrase \( X \) which belongs to \((u,v))\), \( f(T,X) = \text{freq}(v) \) holds.

[Proof] \( X \) occurs at least \( \text{freq}(v) \) times in \( T \) because \( X \) is a prefix of \( \langle v \rangle \). We here assume that \( f(T,X) > \text{freq}(v) \). Then, there exists a subphrase \( Y \) of \( T \) that \( X \) is a prefix of \( Y \) but \( \langle v \rangle \) is not. There also exists a leaf \( w \in V \) that \( Y \) is a prefix of \( \langle w \rangle \). Then, \( \text{TWST}(T,N) \) has the common ancestor \( p \) to \( v \) and \( w \). Since \( X \) is a prefix of
Y, p must be located between the virtual node X and the real node v. However, such real p ∈ V can not exist because u and v are connected by an edge. It contradicts our assumption. Thus we have f(T, X) = \text{freq}(v).

**Lemma 15** Let N > 0 be an integer, T be a text, and TWST(T, N) be the word N-gram tree for T. Then, for any node v ∈ V and v’s parent u, \text{freq}(u) > \text{freq}(v) holds if |⟨u⟩|_# < |⟨v⟩|_#.

[Proof] Since ⟨u⟩ is a prefix of ⟨v⟩, we have \text{freq}(u) ≥ \text{freq}(v). By definition of word N-gram trees, u has another child w and \text{freq}(w) > 0 must hold. Then, we have \text{freq}(v) < \text{freq}(v) + \text{freq}(w) ≤ \text{freq}(v).

Figure 5.2 shows the word 2-gram tree for T = AB#BC#AC#BC#$#, where the number in each node represents the associated frequency \text{freq}(v). The word N-gram tree for any text is isomorphic to the word suffix tree[21] if N = ∞.

## 5.3 Proposed Method

**Keyword Extraction Using Word N-gram Trees**

We describe an algorithm for computing the top-k candidates by using the word N-gram trees. The following two lemmas are important for the correctness of our algorithm.

**Lemma 16** Let N be an integer, T = w_1 · · · w_m be an input text, and V’ be the subset of the word N-gram tree for T such that V’ = \{v ∈ V||⟨v⟩|_# > |⟨u⟩|_#\}, where u is the parent of v. Then, for any candidate W ∈ K(T, N), there exists the node v ∈ V’ such that W = ⟨v⟩.

[Proof] Any candidate W = w_i · · · w_j is either the representing substring of a node or a substring which belongs to a node. In the latter case, let v ∈ V be the node that W
Figure 5.1: An example of computation of a phrase score.

<table>
<thead>
<tr>
<th>phrase</th>
<th>extracting</th>
<th>keywords</th>
<th>from</th>
<th>Web</th>
<th>pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>part-of-speech</td>
<td>verb</td>
<td>noun</td>
<td>other</td>
<td>noun</td>
<td>noun</td>
</tr>
<tr>
<td>PoS score</td>
<td>2</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>word length</td>
<td>11</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>word score</td>
<td>22</td>
<td>80</td>
<td>0</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>phrase score</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>182</td>
</tr>
</tbody>
</table>

$T = AB\#BC\#AC\#BC\#S\#$, $N = 2$

```
root

\[ \infty \]

```

Figure 5.2: The word 2-gram tree of a text $T = AB\#BC\#AC\#BC\#S\#$. 
belongs to \( v \). Then, \( f(T, W) = \text{freq}(v) \) and \( W \) is typified by \( \langle v \rangle \). By definition of the candidate set, \( W \) is not in the candidate set \( K(T, N) \) and it contradicts our assumption. Therefore \( W \) must be represented by a node \( v \in V \). Now we assume that no node in \( V' \) represents \( W \). Then, there exists a node \( u \not\in V' \) such that \( \langle u \rangle = W \). Since \( u \not\in V' \), \( u \) has an ancestor \( v' \in V' \) such that \( |\langle u \rangle|_\# > |\langle v' \rangle|_\# \) and \( f(T, \langle u \rangle) = f(T, \langle v' \rangle) \). Since the representing substring of any candidate must consist of at most \( N \) words, \( |\langle u \rangle|_\# = N \) and no label the path from \( u \) to \( v \) contains \( \# \) in. Then, the representing substring must be \( \langle u \rangle = \langle v \rangle \) and it contradicts our assumption. Therefore any candidate \( W \in K(T, N) \) is represented by a node \( v \in V' \).

\[ \square \]

**Lemma 17** Let \( T = w_1 \cdots w_m \) be an input text and \( N \) be an integer and \( V' \) be the subset of the word \( N \)-gram tree for \( T \) such that \( V' = \{ v \in V ||\langle v \rangle|_\# > |\langle u \rangle|_\# \} \), where \( u \) is the parent of \( v \). Then, for any node \( v \in V' \), there exists a candidate \( W \in K(T, N) \) such that \( \langle v \rangle = W \).

[Proof] Assume that \( \langle v \rangle \not\in K(T, N) \). Since \( v \) represents a word sequence which occurs in \( T \), there exists a candidate \( X \) such that \( \langle v \rangle \) is a prefix of \( X \), \( |\langle v \rangle|_\# < |X|_\# \) and \( \text{freq}(v) = f(T, X) \). Then, there exists a node \( p \) which represents \( X \). However, by Lemma 15, \( \text{freq}(v) > \text{freq}(p) \) holds if \( |\langle v \rangle|_\# < |\langle p \rangle|_\# \). Therefore, for any node \( v \in V' \), \( \langle v \rangle \in K(T, N) \) holds .

To compute the scores for candidates efficiently, we adopt additional information \( \text{WordCount}(v) \) and \( \text{PhraseScore}(v) \) for each node \( v \in V' \), where \( \text{WordCount}(v)|\langle v \rangle|_\# \) and \( \text{PhraseScore}(v) = \text{P}(\langle v \rangle) \), respectively. Figure 5.3 shows the procedure **KeywordExtraction** which computes the scores for all candidates. This procedure first constructs the word \( N \)-gram tree for \( T \). Then, it traverses all the nodes by a depth-first search and compute the score \( PF(X) = \text{PhraseScore}(v) \cdot \log(\text{freq}(v)) \) for each node \( v \in V' \). To sort the candidates by a decreasing order of the score, we store the candidates to a
min-heap. If the heap contains $k + 1$ elements, we remove the candidate that has the minimum score.

Now we can show the following theorem, which is the main result of this chapter.

**Theorem 9** Given an input text $T = w_1 \cdots w_m$, integers $N$ and $k$ and the word $N$-gram tree $N$-TWST($T$), the procedure KeywordExtraction shown in Fig. 5.3 correctly solves the problem in $O(m \log k)$ time.

[Proof] The correctness immediately holds by Lemma 16 and 17.

We then consider the time complexity. Line 1 takes $O(1)$ time for the initialization and line 10 takes $O(1)$ time. The total number of executions of lines 2 to 9 is $O(m)$ times because $N$-TWST($T$) has $O(m)$ nodes by Theorem 8. Lines 3 to 5 take $O(1)$ time in total. Using a min-heap for implementing the set $L$, line 6 takes $O(\log k)$ time for each insertion and line 7 takes $O(\log k)$ time for each deletion. Thus, Lines 2 to 9 take $O(m \log k)$ time in total. Overall, the procedure KeywordExtraction takes $O(m \log k)$ time in total. □

The left column of Table 5.2 shows an example of extracted keywords for a Japanese novel "Rashomon", written by Ryunosuke Akutagawa. Since the input text is written in Japanese, there are no delimiters between the words. Therefore, we used MeCab0.96 for morphological analysis and then inserted white spaces between words. The part-of-speech scores were the same as Table 5.1. We adjusted the lengths of Kanji, Hiragana, and Katakana as same as 3.

**Filtering results**

Although the candidate set excludes some substrings in $T$, some keywords may have common words in themselves, as shown in the left column of Table 5.2. In this keyword list, the word "deshi" (desciple in English) appears six times in the 20 keywords. It
Procedure KeywordExtraction;

Input: a text $T$ of length $n$, an integer $N > 0$,
the word $N$-gram tree $N$-WST($T$), and an integer $k > 0$;

Output: The top-$k$ keywords in $T$;

1. $L = \emptyset$;
2. Traverse $WST(T, N)$ by a depth-first search and do the following:
3. \quad $v =$ the current node and $u = v$’s parent;
4. \quad \textbf{if} $\text{WordCount}(v) > \text{WordCount}(u)$ \textbf{then}
5. \quad \quad $PF(v) = \text{PhraseScore}(v) \cdot \log(freq(v))$;
6. \quad \quad $L = (L \cup \{v\})$;
7. \quad \quad \textbf{if} $|L| > k$ \textbf{then} $L = L \setminus \{x \in L | x$ has the smallest score};
8. \quad \quad \textbf{end if}
9. \quad \textbf{end if}
10. \quad \textbf{return} $L$;

Figure 5.3: The procedure for keyword extraction.
means that another important keyword may be arrested by a few words. Thus, we propose a heuristic approach to reduce such words, called \textit{selection of keywords}.

To obtain more organized keyword lists, we reduce redundant words. Let $L$ be a set of words. $L$ is empty at first. For each keyword $X = w_i \cdots w_j$ in the keyword list in a decreasing order of the scores, we compute the set $\langle L \rangle_X = \{ w_l | i \leq l \leq j, w_l \notin L \}$. Then, for given threshold values $\delta_1, \delta_2 (0 < \delta_1, \delta_2 < 1)$, we accept $X$ and append $\langle L \rangle_X$ to $L$ if
\[
\frac{|\langle L \rangle_X|}{|X|} \geq \delta_1
\]
and
\[
\frac{\sum_{w \in \langle L \rangle_X} W(w)}{\sum_{h=1}^{j} W(w_h)} \geq \delta_2.
\]

The right column in Table 5.2 shows the selected top-20 keywords from the extracted keywords by our proposed algorithm. Compared with the original extracted keywords, there is abundant variety of keyword in the selected keywords. If $k$ is a very small value, threshold values $\delta_1$ and $\delta_2$ should be set to close to 1 and then redundant words will be reduced.

\section*{Keyword extraction from blogs}

User Created Contents, such as blogs, are often written in a colloquial style and heavily biased by their interests. In this subsection, we study a keyword extraction from blogs.

We chose two blogs as the top two results of Google blog search \footnote{http://blogsearch.google.co.jp/} with a search query “Wii Fit”\footnote{http://www.nintendo.co.jp/wii/rfnj/index.html}. We call them blog A and blog B, respectively. Then, for each blog, we extracted of 50 latest posts and created the input data by concatenating the text parts of them. The input texts are 4821 characters for blog A and 1606 characters for blog B.
Table 5.2: The top-20 non-selected and selected keywords extracted from a novel “Hana”, written by Ryunosuke Akutagawa. Since the text is written in Japanese, we translated the keywords to English.

<table>
<thead>
<tr>
<th>Non-selected keywords</th>
<th>Selected keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>弟子の僧 (disciple monk)</td>
<td>弟子の僧 (disciple monk)</td>
</tr>
<tr>
<td>弟子の僧の (of the disciple monk)</td>
<td>内供 (Naigu [personal name])</td>
</tr>
<tr>
<td>内供 (Naigu [personal name])</td>
<td>鼻 (nose)</td>
</tr>
<tr>
<td>鼻 (nose)</td>
<td>禅智内供 (Zenchi Naigu [personal name])</td>
</tr>
<tr>
<td>供 (Gu [fragment of ”Naigu”])</td>
<td>自分 (self)</td>
</tr>
<tr>
<td>弟子 (disciple)</td>
<td>池の尾 (Ikenoo [geographical name])</td>
</tr>
<tr>
<td>弟子の (of the disciple)</td>
<td>顔 (face)</td>
</tr>
<tr>
<td>弟子の僧は (disciple monk is)</td>
<td>中童子 (temple pageboy)</td>
</tr>
<tr>
<td>内供は (Naigu is)</td>
<td>上唇の上から顔の下まで (from the top of his epichile to the bottom of his jowl)</td>
</tr>
<tr>
<td>供は (Gu is)</td>
<td>木の片 (a piece of a tree)</td>
</tr>
<tr>
<td>鼻を ([some transitive verb] one’s nose)</td>
<td>法 (law)</td>
</tr>
<tr>
<td>弟子の僧は (disciple monk is)</td>
<td>寺 (temple)</td>
</tr>
<tr>
<td>禅智内供 (Zenchi Naigu [personal name])</td>
<td>湯 (hot water)</td>
</tr>
<tr>
<td>自分 (self)</td>
<td>眼 (eye)</td>
</tr>
<tr>
<td>池の尾 (Ikenoo [geographical name])</td>
<td>手 (hand)</td>
</tr>
<tr>
<td>僧 (monk)</td>
<td>鏡 (mirror)</td>
</tr>
<tr>
<td>の僧 (the monk of)</td>
<td>心もち (slightly)</td>
</tr>
<tr>
<td>鼻の (of one’s nose)</td>
<td>自尊心 (self-respect)</td>
</tr>
<tr>
<td>内供の (of Naigu)</td>
<td>鼻を粥の中へ落した (someone dropped him/her nose to rice gruel)</td>
</tr>
<tr>
<td>供の (of Gu)</td>
<td>気に (feel)</td>
</tr>
</tbody>
</table>
Table 5.3 shows the top-10 keywords with selection. However, there are undesirable keywords in blog B, because they include some prepositions at their suffixes. The reason for this result is that our algorithm considers any consecutive sequence of words. To fix these results, we may consider another post-processing algorithm based on grammers.

**Utilizing browsing histories**

Some Web pages contain very few text contents. For instance, the extracted posts in blog B in subsection 5.3 contains only 32 characters on average. In this case, statistical information, such as frequency, is not enough reliable.

For this reason, we utilize the browsing histories for users, which is a list of Web pages that the users have seen recently. In this approach, we assume that there are certain Web pages in the browsing histories which are closely related with the current Web page. Then, we consider that a keyword is important if it frequently appears in the browsing histories.

We describe our approach more precisely. Let $T$ be an input text that the user is now browsing. The browsing history $D$ is a collection of strings which contains the input texts created by the Web pages that the user have seen in past times. We assume that the size of $D$ be a fixed size. For any candidate $W$ of keyword for $T$, $f(D, W)$ denotes the total number of the frequencies of $W$ in any input text in $D$. Then, we define the corrected frequency of $W$ in $T$ with $D$ as follows:

$$f'(T, W, D) = f(T, W) \cdot \{1 + \log(f(D, W))\}.$$ 

Next, we consider another problem in the candidate set. Let $XY$ be a phrase, where $X$ is a noun and $Y$ is a preposition. If the input text $T$ is enough large and $X$ is an important word, we expect that $X$ frequently appears in $T$ and is followed by several prepositions. In this case, $X$ will be in the candidate set. However, if $T$ is a short text, such as blog B in Section 5.3, some keywords may appear just once even though they
Table 5.3: An example for extracted keywords from blogs.

<table>
<thead>
<tr>
<th>blog A</th>
<th>blog B</th>
</tr>
</thead>
<tbody>
<tr>
<td>バランスゲーム (balance game)</td>
<td>筋トレ (muscle training)</td>
</tr>
<tr>
<td>踏み台リズム (step ladder-rhythm)</td>
<td>バランス年齢 (balance age)</td>
</tr>
<tr>
<td>Wii Fit (Wii Fit)</td>
<td>体重 (weight)</td>
</tr>
<tr>
<td>ヘディング (Heading)</td>
<td>データ (data)</td>
</tr>
<tr>
<td>有酸素運動 (aerobic exercise)</td>
<td>－ (the dash symbol)</td>
</tr>
<tr>
<td>リズムボクシング (rhythm-boxing)</td>
<td>ながらマラソン (marathon with doing something)</td>
</tr>
<tr>
<td>バランススノボー (balance monoboarding)</td>
<td>腹筋も (also abdominal muscle)</td>
</tr>
<tr>
<td>スキー skiing</td>
<td>画面の (of the screen)</td>
</tr>
<tr>
<td>バランス年齢 balance age</td>
<td>ウィーボ (Weebo [character name])</td>
</tr>
<tr>
<td>体重 (weight)</td>
<td>休み (holiday)</td>
</tr>
</tbody>
</table>
are important.

We utilize the browsing histories for recognizing the positions that should be cut. Let $v \in V$ be any leaf whose frequency $f(v)$ is one. Then, we substitute the longest prefix $p_d(\langle v \rangle)$ which belongs to the ingoing edge to $v$ and also appears in $D$ for the original candidate $\langle v \rangle$. In other words, $p_d(\langle v \rangle)$ is a candidate of keyword for $D$ and we also choose $p_d(\langle v \rangle)$ as a candidate for $T$ if $p_d(\langle v \rangle)$ appears in $T$.

An example of keyword extraction from blogs

As an example for our approach, we show a result of a keyword extraction from a short input text.

We assume that the browsing history is blog A or blog B in Section 5.3. The input text is chosen from a search result of the search query “Wii Fit” by Google blog search. The has 553 characters in total. We call the input text “blog C”.

Table 5.4 shows the result of the top-10 keywords from blog C by our previous algorithm, the keywords using blog A as the browsing history, and the keywords using blog B as the browsing history.

The extracted keywords by using blog A or blog B contain the phrase “muscle training” and “aerobic exercise”, which can be considered as an important keyword in blog posts about Wii Fit. Thus our approach seems to work well even though they appear just once in blog C for each.

However, there are still undesirable keyword, such as “do” or “like”. The scores for them become high because these general term frequently appear in any document. To reduce the side effects, applying TF-IDF[41] to word scores is a possible candidate.
Table 5.4: Keyword extraction from blog C with or without a history.

<table>
<thead>
<tr>
<th>without history</th>
<th>using blog A</th>
<th>using blog B</th>
</tr>
</thead>
<tbody>
<tr>
<td>ジョギング (jogging)</td>
<td>ジョギング (jogging)</td>
<td>ジョギング (jogging)</td>
</tr>
<tr>
<td>ペース (pace)</td>
<td>Wii Fit (Wii Fit)</td>
<td>有酸素運動 (aerobic exercise)</td>
</tr>
<tr>
<td>気に入った (like)</td>
<td>有酸素運動 (aerobic exercise)</td>
<td>筋トレ (muscle training)</td>
</tr>
<tr>
<td>Wii Fit (Wii Fit)</td>
<td>筋トレ (muscle training)</td>
<td>ペース (pace)</td>
</tr>
<tr>
<td>島 (island)</td>
<td>バランスボード (balance board)</td>
<td>気に入った (like)</td>
</tr>
<tr>
<td>する (do)</td>
<td>感じ (feel)</td>
<td>Wii Fit (Wii Fit)</td>
</tr>
<tr>
<td>ないので (because it is not)</td>
<td>ペース (pace)</td>
<td>する (do)</td>
</tr>
<tr>
<td>ました (did)</td>
<td>気に入っ (like)</td>
<td>島 (island)</td>
</tr>
<tr>
<td>し (do)</td>
<td>する (do)</td>
<td>バランス (balance)</td>
</tr>
<tr>
<td>です (is)</td>
<td>ヨガ (yoga)</td>
<td>画面 (screen)</td>
</tr>
</tbody>
</table>
5.4 Experimental results

We implemented our keyword extraction algorithm in C++ and compiled by Visual C++ .NET 2005. We ran the program on a system with Pentium M 1.3GHz, 1GB RAM, and Windows XP.

Dataset

We corrected the input text $T$ from Aozora Bunko $^3$. The text was the novel “Genji Monogatari”, written by Akiko Yosano. We removed the rubies, left a space between words by Mecab 0.96, and then converted to a UTF-8 text. The input text $T$ consists of about 0.88 million Japanese characters or about 3 million bytes. Since our program regards each byte as a character, the size of $T$ is defined as the number of bytes.

Method

Experiment 1

For any integer $1 \leq N \leq 20$, we constructed the word $N$-gram tree for $T$ and measured the number of nodes, computation time for the construction, and computation time for keyword extraction. The number of extracted keywords was 30. Another settings were the same as an example of Table 5.2.

Experiment 2

We measured the speed against the size of the input text. We set $N = \infty$. Another setting were the same as Experiment 1.

$^3$http://www.aozora.gr.jp/
Results

Result 1

Figure 5.4 shows the result of node counts and Fig. 5.5 shows the result of computation time.

As shown in Fig. 5.4, the node count increased with $N$. The node counts were the same as $\infty$-$WST(T)$ when $N \geq 18$. Similar to the result of the word-based truncated suffix trees in chapter 3, the node count was about a half of $\infty$-$WST(T)$ when $N = 3$.

The computation time also increased with $N$ in the result shown in Fig. 5.5. When $N = 8$, the computation time were almost the same. Most computation time for the whole keyword extraction from each input text was accounted for the construction of the word $N$-gram tree.

As mentioned above, the word $N$-gram tree can save time and space when a small $N$ is chosen. On the other hand, our algorithm also extract keywords in a reasonable time even when $N = \infty$. In our implementation, each node takes 28 bytes. Thus, the space consumption is about 1.4 bytes/character, 3.8 bytes/character, 8.1 bytes/character for 2, 3, $\infty$, respectively.

Result 2

Figure 5.6 shows the result. As shown in the figure, our algorithm seem to run in a proportional time to the text length. The total computation time for the whole text is about 5.5 sec, including morphological analysis of the Japanese text, construction of the word $N$-gram tree, and our keyword extraction algorithm. In general, Web pages contain more small texts, therefore we conclude that our keyword extraction algorithm can be used in real-time Web browsing.
Figure 5.4: Node count vs maximal word count $N$. 

78
Figure 5.5: Computation time vs maximal word count $N$. 
Figure 5.6: Computation time vs input size.
5.5 Discussion

In this chapter, we proposed the word N-gram tree, which is an extension of both the word-based truncated suffix tree and the word suffix tree. We also considered a keyword extraction problem for supporting Web browsing. Using the word N-gram tree, we proposed a keyword extraction algorithm that runs in $O(n \log k)$ time, where $n$ is the length of the input text and $k$ is the number of extracting keywords.
Detecting and removing spam messages is a matter of considerable concern to everybody. Spams are ubiquitous in the diversified media of the Internet and we suffer great losses from them; more than 90 percent of all emails have been reported as spam [11], and more than 75 percent of all blog entries have also been reported as splog (spam-blog) [30]. They wastefully consume network resources, unnecessarily lay a burden on server systems, and degrade the accuracy of search results. According to a report of European Commission [39], spam emails cause damage worths 39 billion euros worldwide. Some countries have placed legal restraints on sending spam messages, but we can not say that these restraints contribute to decrease the number of spam messages. Therefore, it is essential to develop an efficient method that detects spam automatically.

A lot of automatic spam detection methods have been proposed. In the early stages of the history, rule-based methods were widely used, such as compilation blacklists of IP addresses. Then machine-learning-based methods, such as a Bayesian method in [18], have been used; these methods learn probabilistic distributions of some features from given training examples and then judge a new coming email whether it is spam or
non-spam, say *ham*, using the learned distributions. Since it is too costly to create the necessary training data, unsupervised methods have been proposed [38, 37, 57]. However, to circumvent these methods, spammers have created spam messages in more sophisticated ways, such as *word salads*. We have to develop a novel method for such a new type of spam.

Spam detection methods are categorized into two groups: non-content-based methods and content-based ones. A blacklist is a typical example of the former, and the Bayesian method in [18] is the latter. The algorithms stated in [38, 37], which find characteristic substrings in the input documents to recognize spam, are content-based methods, too. The algorithm in [57] judges some emails as spam if the number of similar emails is larger than a given threshold value. The algorithm in [34] estimates language models of spam and ham messages. The algorithms stated in [19, 4] utilize the hyperlink structure of contents. For word salads, there are a few reports. Misra *et al.* [35] present a method that measures the semantic coherence of a document by using Latent Dirichlet allocation. Their idea is based on the premise that a *ham* contains only a few topics but a spam is made up of many topics. Jeong *et al.* [23] present a method that utilizes thesauruses, which regards a message as a spam when the distribution of terms in it differs from the normal distribution of the thesauruses.

In this chapter, we study content-based spam detection methods. We assume that spam documents are automatically generated as copies of a seed document with allowing some random modifications [6], since we can observe that the aim of spammers is to obtain large rewards with a slight effort and hence they should create a large number of copies. Conversely, the *cost* of generating a normal document should be relatively high.

The main contribution of this chapter is to propose an unsupervised algorithm for spam detection, which we call the *spam detection algorithm by Document Complexity Estimation* (DCE for short). The key idea is to measure the cost of generating each
document by an entropy-like measure, called *document complexity*. It is defined as
the ideal average code-word-length of the document by presuming the input collection
excluding the target itself. We expect that the complexity for ham is high, but low
for spam. As the document complexity, however, is an ideal measure like Kolmogorov
complexity, we substitute an estimated occurrence probability of each document for
its complexity. We treat each document as a sequence of letters over a finite alphabet
rather than a bag-of-words, and estimate its generation probability from a latent model
of the input collection by using *Maximal Overlap method* proposed by Jagadish *et
al.* [22].

Maximal Overlap method [22] (**MO** for short) is a technique for estimating the
probability $P(x)$ of a long string $x$ from the probabilities of substrings in $x$. Although
a straightforward method based on the original **MO** takes more than quadratic time to
the target string length, we present an efficient algorithm that runs in linear time to it.
We call it as **DCE by MO** algorithm (**DCE_MO** for short). The keys of the algorithm is
full exploitation of the suffix tree and suffix links.

We ran experiments on the real data from popular bulletin boards and some syn-
thetic data. We compared our algorithm with the algorithm stated in [37], which is also
an unsupervised method. For the real data, the results of **DCE_MO** are comparable to
the algorithm in [37]. Experimental results on the synthetic data showed that **DCE_MO**
particularly works well for spam messages based on random replacement, such as word
salad and noise spam.

The rest of this chapter is organized as follows. In 6.1 we reviews basic notions.
In 6.2 we gives our spam detection algorithm. In 6.3 we reports experimental results.
Finally, in 6.4 we conclude this chapter.

This chapter is based on the paper [47].
6.1 Preliminaries

Strings
Let $\mathbb{N}$ be the set of all non-negative integers, and let $\Sigma$ be a finite alphabet. We denote by $\Sigma^*$ the set of all finite strings over $\Sigma$. The length of $x$ is denoted by $|x| = n$. The empty string is the string of length zero and denoted by $\varepsilon$. For a string $s$, if $s = xyz \in \Sigma^*$ for some strings $x, y, z \in \Sigma^*$, then $x, y,$ and $z$ are called a prefix, a substring, and a suffix of $s$, respectively. If $s = a_1 \cdots a_n \in \Sigma^*$ ($a_i \in \Sigma$) is a string of length $n$, then for every $1 \leq i \leq n$, the $i$-th letter of $s$ is denoted by $s[i] = a_i$, and the substring from $i$ to $j$ is denoted by $s[ i..j ] = a_i \cdots a_j$; we define $s[ i..j ] = \varepsilon$ if $i > j$ for convenience. The concatenation of strings $s$ and $t$ is denoted by $s \cdot t$, or simply $st$.

We say that a string $s$ occurs in string $t$ at the position $i$ if $s[1..|s|] = t[i..i + |s| - 1]$.

For a set $D = \{ s_1, \ldots, s_M \} \subseteq \Sigma^*$ ($M \geq 0$) of $M$ strings, we define $|D| = M$ and $||D|| = \sum_{s \in D} |s|$. We define the sets of all substrings and all suffices of $D$ by $\text{Sub}(D) = \{ s[i..j] : s \in D, 1 \leq i, j \leq |s| \}$ and $\text{Suf}(D) = \{ s[i..|s|] : s \in D, 1 \leq i \leq |s| \}$, respectively.

Suffix Trees
Let $D = \{ s_1, \ldots, s_M \} (M \geq 0)$ be a set of $M$ strings over $\Sigma$ with the total size $N = ||D||$. The suffix tree for $D$, denoted by $\text{ST}(D)$, is a compacted trie $\text{ST}(D) = (V, E, \text{root}, \text{suf}, \text{lab}, \text{fr})$ [54] for the sets of all suffices of the strings in $D$ as shown in 6.1.

Formally, the suffix tree for $D$ is a rooted tree, where $V$ is the vertex set, $E \subseteq V^2$ is the edge set, $\text{suf} : V \to V$ is a suffix link, $\text{lab} : E \to \Sigma^*$ is an edge-labeling, and $\text{fr} : V \to \mathbb{N}$ is a frequency counter. All the out-going edges leaving from a vertex always start with mutually different letters. Each vertex $v \in V$ represents the unique substring $\alpha = \langle v \rangle \in \text{Sub}(D)$, where $\langle v \rangle$ is the concatenation of the labels on the unique path from the root to $v$. Therefore, we identify a vertex with the corresponding string
Fig. 6.1: The suffix tree for $D = \{cacao\$, oca\#\}$. Broken lines represent the suffix links. Figures in circles indicate the frequency counters.

hereafter. For every internal vertex $[ca]$ starting with a symbol $c \in \Sigma$, there always exists a suffix link from $[ca]$ to the unique vertex $suf([ca]) = [\alpha]$. Finally, the leaves of $ST(D)$ represent the set $Suf(D)$, and thus, $ST(D)$ stores $Sub(D)$. In 6.1, we show the suffix tree for a string $s = cacao\$, where solid and broken lines represent the edges and the suffix links, respectively. Numbers in vertices represent their frequency counters. We note that $\$ is the end-marker which does not occur in $s[1..|s| − 1]$, and then $ST(s)$ strictly has $|s|$ leaves.

$ST(D)$ has at most $2N − 1$ vertices since the common prefix of paths can be shared. Ukkonen [54] presented an elegant algorithm that builds $ST(D)$ in $O(N)$ time by traversing the tree using suffix links. We augment each vertex $v = [\alpha]$ with the counter $fr(v) \in \mathbb{N}$, which keeps the frequency of $\alpha$ as a substring in strings of $D$. We can see that $fr([\alpha])$ equals the number of leaves that are descendants of $[\alpha]$. 

87
Blog and Bulletin Board Spam-Generation Process

We focus on a type of spam called blog and bulletin board spam. 6.2 shows an outline of a spam-generating mechanism for this type of spam.

We first assume a large collection $D_0$ consisting of normal documents randomly drawn from the document source $D$. In typical situations, normal documents are posted by human users. Next we assume a document $s_0$ is drawn randomly as a spam seed from the spam document source $S$ to generate a collection of spam documents.

For an integer $K \geq 1$, which is unknown to the users, $K$ approximate copies $s_1, \ldots, s_K$ of the seed $s_0$ are generated with allowing the following perturbations:

**Mutation** Changing randomly selected letters in a document.

**Crossover** Exchanging randomly selected parts of a pair of documents.

**Word Salad** Replacing a set of randomly selected words in a sentence with randomly selected words drawn from other sentences so that it is still grammatically correct, but meaningless.
Then, the spam documents $s_1, \ldots, s_K$ are added to $D_0$. The input collection of documents $D = (d_1, \ldots, d_n)$, $n \geq 0$ is generated by repeating this process for different spam seeds. Note that we can not see in advance how many spam seeds are selected and how many spam documents are generated. A document $d \in D$ is said to be dirty if it is spam, and clean if it belongs to the original collection $D_0$.

We note that our detection algorithm in 6.2 is adaptive, and thus it does not need a priori knowledge about the document sources $D$ and $S$, classes of modifications, and the number $K$ of approximate copies per seed.

**Spam Detection Problem**

Given an input collection $D$, we consider the problem of classifying all documents in $D$ into clean or dirty. Formally, our goal is to devise a mapping $f : D \rightarrow \{0, 1\}$ that makes classification, where the values 0 and 1 indicate the states that $d$ is clean or dirty, respectively. We call the mapping decision function. We measure the performance of $f$ on $D$ by some cost functions. Let $M_+$ be the number of correctly detected spam documents, and $M_-$ be the number of normal documents judged as spam. Let also $N_+$ be the total number of spam documents in $D$. Then, the recall is defined by $R = M_+/N_+$ and the precision is defined by $P = M_+/(M_+ + M_-)$. The F-score is also defined by $F = 2PR/(P + R)$.

**6.2 Proposed Method**

**Outline of Proposed Method**

Our method is parameterized by a given probabilistic model $M$ for probability distribution over $\Sigma^*$. We assume the model $M = \{ Q(\cdot \mid \theta) : \theta \in \Theta \}$, which is a family of probability distributions over strings $Q(\cdot \mid \theta) : \Sigma^* \rightarrow [0, 1]$ with parameter $\theta$ drawn
from some parameter space $\Theta$. A parameter $\theta$ can be a description of a non-parametric model, e.g., Markov chains, as well as a parametric model.

The basic idea of our method is as follows. Let $D \subseteq \Sigma^*$ be an input collection of $M$ documents. Let $d \in D$ be any target document in $D$. If we model $D$ by some probability distribution $Q(\cdot \mid \theta)$ for some $\theta \in \Theta$, then we can encode $D$ by a certain encoding whose code length is

$$L(D \mid \theta) \simeq \lceil -\log Q(D \mid \theta) \rceil$$

so that the best compression ratio can be achieved [13]. In what follows, we denote by $\theta_D$ this best parameter $\theta$ for a given collection $D$ within $\Theta$. If $d$ is dirty, then $d$ is a copy of some spam seed $s_0 \in S$ that has a certain number of copies in $D$. In this case, we expect that $d$ is statistically similar to some portion of $D$, and thus, the contribution of $d$ to the whole $D$ will be very small.

On the other hand, we suppose that $D$ is obtained from the collection $D' = D \setminus \{d\}$ of $M - 1$ documents by adding the document $d$. $D'$ can again be encoded by some $\theta' = \theta_{D'} \in \Theta$ in length $L(D' \mid \theta')$. Then, the contribution of $d$ can be measured by how much this addition of $d$ increases the code length of $D$, which is bounded above by

$$\Delta L(D, d) \overset{\text{def}}{=} L(D' \cup \{d\} \mid \theta) - L(D' \mid \theta') = L(D \mid \theta) - L(D' \mid \theta') \leq L(d \mid \theta'),$$

where $\theta = \theta_D$ and $\theta' = \theta_{D'}$. In the above equations, the inequality in the last row follows from the following observation: A description of $D$ can be obtained by appending to the description for $D'$ of length $L(D' \mid \theta')$ an encoding for $d$ of length $L(d \mid \theta')$ for $d$ conditioned by $D' = D \setminus \{d\}$. Therefore, we have an inequality $L(D \mid \theta) \leq L(D' \mid \theta') + L(d \mid \theta')$.

We expect that if the target document $d$ has more copies in $D$ then the contribution $\Delta L(D, d)$ will be smaller and thus more likely to be spam. Then, $\Delta L(D, d)$ can be
estimated by the upperbound $\Delta L(D, d) \simeq L(d \mid \theta') \simeq \lceil -\log Q(d \mid \theta') \rceil$. This is our decision procedure for spam detection:

$$f(d) = \begin{cases} 1 & \text{if } \lceil -\log Q(d \mid \theta') \rceil / |d| \leq \gamma \\ 0 & \text{otherwise,} \end{cases} \quad (6.1)$$

where $\gamma > 0$ is a threshold parameter, which will be given in the following section. We define the document complexity of $d$ for $D$ by $\ell = \lceil -\log Q(d \mid \theta') \rceil / |d|$.

### Document Probability Estimation by MO Method

We did not mention about the concrete model that the probability of each document follows. In general, to compute the precise value of $Q(d \mid \theta')$, namely the document complexity of $d$ for $D$, is quite hard. Therefore, we substitute an estimated occurrence probability of $d$ for its complexity. We denote the substitute probability by $\tilde{Q}(d \mid \theta')$.

We use Maximal Overlap method [22] (MO for short) to estimate $\tilde{Q}(d \mid \theta')$.

MO estimates the probability $P(x)$ of a long string $x \in \Sigma^*$ based on a set of shorter substrings appearing in the original string in $D$ as follows. Given string $\beta$, the conditional probability for a string $\alpha$ is given by $P(\beta \mid \alpha) = P(\alpha\beta)/P(\alpha)$. Let $s \in \Sigma^*$ be a target string to estimate $\tilde{Q}(s \mid \theta_D)$. In general, for any $\tilde{Q}(\cdot \mid \theta)$, we can write the probability of the string $s = \beta_1 \cdots \beta_m$ ($m \geq 1$) as $P(s) = P(\beta_1) \prod_{i=2}^{m} P(\beta_i \mid \beta_1 \cdots \beta_{i-1})$. For each $i = 1, \ldots, m$, MO approximates the conditional probability $P(\beta_i \mid \alpha_i)$ with the unique longest suffix $\alpha_i$ of $\beta_1 \cdots \beta_{i-1}$, called the longest context, such that $\alpha_i \beta_i$ appears in $D$. For substrings $u$ and $v$ of an input string $s$, we define the maximal overlap, denoted by $u \otimes v$, as the maximal string $w \in \Sigma^*$ that is both a suffix of $u$ and a prefix of $v$.

Recall that $\text{Sub}(D)$ is the set of substrings in $D$. In 6.3, we show the original MO method that computes an estimation of $P(s)$ in the greedy way. For every $i = 1, \ldots, |s|$ ($|s| \geq 1$), MO finds maximally overlapping substrings $\gamma_i = \alpha_i \beta_i \in \text{Sub(\{s\})}$.
Algorithm MO:

*Input:* Any data structure $Sub(D)$ for a set $D \subseteq \Sigma^*$ of input strings, frequency counter $fr : Sub(D) \rightarrow \mathbb{N}$, and a string $s \in \Sigma^*$ of length $L$.

*Output:* An estimated probability of $\tilde{Q}(s|\theta_D)$.

(i) Let $\gamma_0 = \varepsilon$ be the context, $fr(\varepsilon) = ||D||$, $\theta_D = (Sub(D), fr)$, and $i := 2$.

(ii) While $s \neq \varepsilon$ holds, we repeat the following process: Find the unique longest substring $\gamma_i \in Sub(D)$ that maximizes $\gamma_i \otimes \gamma_i$ such that $\gamma_i \otimes s \neq \varepsilon$. If there are more than two substrings with the same overlap, then take the longer one. In case that there exists no such $\gamma_i$, set $\alpha_i = \varepsilon$, $\beta_i = \gamma_i$ to be the initial letter of $s$, $P(\beta_i | \alpha_i) = 1/||D||$. Let $\alpha_i = \gamma_{i-1} \otimes \gamma_i$ and $\beta_i = \gamma_i \otimes s$. Remove the overlap $\gamma_i \otimes s$ from $s$. Define $P(\beta_i | \alpha_i) = fr(\alpha_i \beta_i)/fr(\alpha_i)$.

(iii) Set $m = i$.

Return $\tilde{Q}(s | \theta_D) = P(\beta_1) \prod_{i=2}^m P(\beta_i | \alpha_i)$.

Figure 6.3: The original MO algorithm for estimating the probability $Q(s | \theta_D)$ such that $\beta_i \neq \varepsilon$. Therefore, we define the associated probabilities by $P(\beta_i | \alpha_i) = fr(\alpha_i \beta_i)/fr(\alpha_i)$. If there exists no such $\gamma_i$ then the next letter $s[k] \in \Sigma$ has never appeared in $D$, where $k = |\beta_1 \cdots \beta_{i-1}|$. Then in this zero-probability case, we set $\alpha_i = \varepsilon, \gamma_i = \beta_i$ to be the un-seen letter $s[k] \in \Sigma$, and define $P(\beta_i | \alpha_i) = 1/N$, where $N = ||D||$ is the total letter frequency. Note that $s = \beta_1 \cdots \beta_{|s|}$. Finally, we compute the estimated probability by $\tilde{Q}(s | \theta_D) = P(\beta_1) \prod_{i=2}^{|s|} P(\beta_i | \alpha_i)$.

Jagadish et al. [22] show that MO is computable in $O(\ell q)$ time for given the suffix tree $ST(D)$, where $\ell = |s|$ and $q$ is the depth of $ST(D)$. That is, MO takes $O(\ell^2)$ time in worst case. They also show that MO outperforms the previous method KVI [31],
Algorithm LinearMO:

Input: a string $s$, the suffix tree $ST(D)$ for a set of strings $D$.

Output: an estimated probability of $\tilde{Q}(s|\theta_D)$.

1: $v :=$ the root of $ST(D)$;
2: for $i := 1, \ldots, \ell = |s|$ do
3: $x_i := s[i]$;
4: while $v$ has no out-going edge $(v, w)$ labelled with $x_i$ do
5: if $(v = \text{root})$ then $Q := Q \cdot (1/N)$ with $N = ||D||$;
6: else $v := \text{suf}(v)$ by following the suffix link;
7: end while
8: $Q := Q \cdot fr(w)/fr(v)$;
   /* $\alpha_i = \text{str}(v)$ and $P(x_i|\alpha_i) = fr(w)/fr(v)$ */
9: $v := w$ by following the edge;
10: end for
11: return $Q$; /* estimation of $\tilde{Q}(s|\theta_D)$ */

Figure 6.4: A linear time MO-score estimation algorithm
which uses the non-overlapping decomposition of the input string in estimation.

Efficient Computation of MO

In 6.5, we show the outline of the proposed algorithm DCE_MO that runs in $O(N)$
to detect all spam candidates, where $N = ||D||$ is the total length of documents.
The crucial parts of the algorithm are at Line 4 and 5. That is a time-consuming
process since we have to iteratively build the suffix tree $M$ times for $D$ except for
the current document $D$ to estimate $\tilde{Q}(d|\theta')$ for each $d \in D$. Therefore, a straightforward
implementation requires $O(MN + M\ell^2) \simeq O(MN + N\ell) = O(MN)$ time, where
$M = |D|$, $N = ||D||$, and $\ell = \max_{d \in D} |d|$.
Speed-up of MO estimation.

6.4 shows a linear-time algorithm \texttt{LinearMO} for estimating $Q(s|\theta_D)$ using $ST(D)$. The algorithm quickly finds the longest context $\alpha_i$ for each factor $\beta_i$ by traversing the suffix tree using suffix links via a technique similar to [54]. By the next lemma, \texttt{LinearMO} improves computation time of the original MO from $O(M\ell^2)$ time to $O(M\ell) \approx O(N)$ time.

**Lemma 18** Let $\Sigma$ be an alphabet of constant size. The algorithm \texttt{LinearMO} in 6.4 correctly implements the algorithm MO in 6.3 to estimate $Q(s|\theta_D)$, and runs in $O(\ell)$ time and $O(N)$ space, where $N = ||D||$ is the total size of the sample $D$ and $\ell = |s|$ is the length of the string $s$.

[Proof] Let $\gamma_i = \alpha_i \beta_i$ ($m \geq 1, 1 \leq i \leq m$) be the maximally overlapping substrings computed in the original MO. Assume that $\beta_j = x_h \cdots x_i$ ($h \leq i$) with $k = |\beta_i|$. Then, we can show that $P(\beta_i | \alpha_i) = P(x_h | \alpha_i)P(x_{h+1} | \alpha_i x_h) \cdots P(x_i | \alpha_i x_h \cdots x_{i-1})$ since the longest context of $x_j$ in $s$ relative to $ST(D)$ is $\alpha x_h \cdots x_{j-1}$ for each $h \leq j \leq i$. Thus, the correctness follows. The time complexity is derived from arguments similar to [54].

\[ \square \]

Avoiding repetitive construction of suffix trees.

In the building phase at Line 4 of Algorithm \texttt{DCE\_MO}, a straightforward implementation takes $O(MN + M\ell^2)$ time by building the suffix tree $ST(D\{d\})$, called a leave-one-out suffix tree for $D$, for each document $d \in D$. Instead of this, we simulate traversal of the leave-one-out suffix tree $ST(D\{d\})$ directly on $ST(D)$ for $D$ and $ST(\{d\})$. A virtual leave-one-out suffix tree for $D$ and $d$ is a suffix tree $ST(D\{d\}) = (V', E', \text{root}', \text{suffix}', \text{label}', \text{fr}')$ defined by $V' = \{ [\alpha] : \alpha \in \Sigma, fr_D(\alpha) \geq 1, fr_d(\alpha) \geq 1 \}$ and $E' = \{ ([\alpha], [\beta]) \in V' \times V' : ([\alpha], [\beta]) \in E_D \}$, where $fr_D(\alpha)$ and $fr_d(\alpha)$ are the
Algorithm DCE_MO(D):

Input: An input collection \( D \subseteq \mathcal{D} \), a threshold \( \gamma > 0 \).

Output: A set \( A \subseteq D \) of dirty documents in \( D \).

1: Construct the suffix tree \( ST(D) \) for the whole document \( D \);
2: \( A := \emptyset \);
3: foreach \( d \in D \) do
   4: Simulate \( Q := \text{LinearMO}(d, ST(D\{d\})) \) by running copies of LinearMO simultaneously on \( ST(D) \) and \( ST(\{d\}) \);
   5: \( L := -\log Q \);
   6: if \( L/|d| \leq \gamma \) then \( A := A \cup \{d\} \);
      /* \( d \) is detected as a spam */
4: end for
8: return \( A \);

Figure 6.5: An outline of our spam detection algorithm

frequency counters of \( \alpha \) on \( ST(D) \) and \( ST(d) \), respectively. We define the frequency of a vertex \([\alpha]\) by \( fr^*[\alpha] = fr_D([\alpha]) - fr_d([\alpha]) \).

Lemma 19 Let \( D \subseteq \Sigma^* \) and \( d \in \Sigma^* \) be any collection and a document. Suppose that \( d \in D \). Then, \( \overline{ST}(D\{d\}) \) is isomorphic to the original \( ST(D\{d\}) \).

Lemma 20 Let \( D \subseteq \Sigma^* \) be any sample. For any document \( d \in D \) of length \( \ell \), we can simulate the algorithm LinearMO in 6.4 over the virtual suffix tree \( \overline{ST}(D\{d\}) \) by running the copies of LinearMO simultaneously over \( ST(D) \) and \( ST(\{d\}) \) by the above rule. The obtained algorithm runs in \( O(\ell) \) time with preprocess \( O(||D|| + \ell) \) for constructing \( ST(D) \) and \( ST(\{d\}) \).

[Proof] The algorithm has a pair \((v_D, v_d)\) of two pointers \( v_D \in V_D \) and \( v_d \in V_d \), respectively, on vertices of \( ST(D) \) and \( ST(\{d\}) \). When the algorithm receives the
next letter $c = s[i]$, it moves from a combined state $(v_D, v_d)$ to $(v'_D, v'_d)$ such that $v'_D = \lfloor \text{str}_D(v_D) \cdot c \rfloor$ and $v'_d = \lfloor \text{str}_d(v_d) \cdot c \rfloor$ if $\text{fr}_D(v'_D) - \text{fr}_d(v'_d) \geq 1$ holds. For the zero-probability case (at line 5), we set $N := ||D|| - |d|$. To see that the moves are always well-defined, we define the pair for $v$ by $\pi(\alpha) = (\text{sgn} \text{fr}_D(\alpha), \text{sgn} \text{fr}_d(\alpha)) \in \{+,0\} \times \{+,0\}$ for any vertex $v = [\alpha]$ ($\alpha \in \Sigma^*$). Since $d \in D$ and $\alpha \in \text{Sub}(d)$ hold by assumption. Then, we can show that $(+, +)$ is only possible state for $\pi(\alpha)$. Hence, the result immediately follows.

From Lemma 18 and Lemma 20, we show the main theorem of this paper.

**Theorem 10** The algorithm DCE_MO in 6.5 runs in $O(N)$ time and $O(N)$ space, where $N = ||D||$ is the total size of $D$.

**Determining the Threshold**

Our unsupervised spam detection method DCE_MO in 6.5 takes a decision threshold $\gamma > 0$ as a learning parameter. To determine an appropriate value of $\gamma$, we first draw the histogram of the document complexity $L$ over unlabelled examples only, and then adaptively define $\gamma$ to be the point that takes the minimal value in an appropriate range, say, $0.0 \sim 1.0$ (bit).

To justify this selection of $\gamma$, we show in 6.6 the histograms of the document complexity $L$ over all documents in test collection YF4314, which we will state later. In the figure, we see that the distribution of spam documents is concentrated within the range of $L = 0.0 \sim 0.2$ (bit) per letter, while that of normal documents spreads over $L = 1.0 \sim 3.5$ (bit) per letter and obeys a bell-shaped distribution with the peak around 2.2 (bit). Thus, it is a straightforward strategy to minimize classification risk by taking $\gamma$ to be the point that takes the minimal value. This justifies the choice of $\gamma$. 96
6.3 Experimental Results

Dataset

We used a test collection of forums from Yahoo Japan Finance\(^1\), collected by Narisawa et al. [37]. The collection consists of four sections of forum data: YF4314, YF4974, YF6830, and YF8473. All posts of each forum are labelled if they are spam or not. 6.1 shows the details of them. In the following experiments, we used only the body texts of documents, including HTML tags.

Method

For simplicity, we call the proposed algorithm by DCE from now on. We implemented DCE and the following methods.

Naive method: Let \( D \) be the input document set. Then, the Naive method regards a document \( d \) as spam if \( d \) is a substring of another document in \( D \). That is, Naive is a type of copy-detection method.

Alienness measure: Narisawa et al. [37] propose a spam detection method which uses the representatives of substring classes, which are characteristic strings extracted from the document set. There are three measures for representatives, \textit{length}, \textit{size}, and \textit{maximin}, and we call the methods with them the AM\_Len, the AM\_Siz, and the AM\_Max, respectively. They also propose a method for determining the threshold for their measures. Their methods then regard a document as spam if it contains a representative such as alien, which is an outlier of substrings in nominal documents.

\(^1\)http://quote.yahoo.co.jp
Figure 6.6: Histogram of the document complexity of Web data.
Table 6.1: Details of the datasets.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of spam</th>
<th># of ham</th>
<th>Total length</th>
</tr>
</thead>
<tbody>
<tr>
<td>YF4314</td>
<td>291</td>
<td>1424</td>
<td>184,775</td>
</tr>
<tr>
<td>YF4974</td>
<td>331</td>
<td>1315</td>
<td>211,505</td>
</tr>
<tr>
<td>YF6830</td>
<td>317</td>
<td>1613</td>
<td>252,324</td>
</tr>
<tr>
<td>YF8473</td>
<td>264</td>
<td>1597</td>
<td>239,756</td>
</tr>
</tbody>
</table>

Results

Exp. 1: Computational efficiency.

First, we compared the implementation of LinearMO with a straightforward implementation of MO. The input document set \( D \) for this experiment is a concatenation of all the datasets. The total length \( \|D\| \) is 888360. 6.7 shows the computation time for the whole spam detection and for the probability estimation. As shown in the graph, the computation time for LinearMO is more than four times faster than the straightforward implementation. On the other hand, the total computation time is at most twice faster than that because the construction for suffix trees takes much time. The memory space consumption of the suffix tree of the whole \( D \) is about 29.7MB, including \( D \) itself.

Exp. 2: Basic Performance.

6.2 shows the recalls, precisions, and F-scores for all methods. As shown in the table, DCE and AM_Siz achieved a high recall, DCE and Naive achieved high precision, and DCE achieved the highest F-score in all the datasets. Overall, DCE shows slightly better performance compared to other methods.
Figure 6.7: Computation time.
Exp. 3: Recall and Precision.

In this experiment, we did not use the methods to determine thresholds of each method. Instead, we output all documents as spam by ascending order of its document complexity and computed recalls and precisions. The graph in 6.8 shows the recall and the precision curves of the dataset YF6830. To the left of the intersection of the precision and the recall of DCE, precision and recall are clearly higher than for any other methods. However, to the right of the intersection, the precision decreases faster than for the other methods.

Exp. 4: Performance Against Noise Density.

We created noise spam by using a seed as follows: For each position $1 \leq i \leq |d|$ of $d$, we replaced $d[i]$ by $d[j]$ with probability $p$, where $1 \leq j \leq |d|$ is a random number. We call the probability the density of noise. To examine the effect of the density, we set the number of spams to 20. 6.9 shows the recall rates against density. In Exp. 4 and 5, we redefine the recall by (detected noise spams / inserted noise spams). AM_Siz was the best in the five algorithms. DCE were comparable to AM_Siz when $p \leq 0.005$. However, its recall decreased depending on the increase of the density.

Exp. 5: Performance Against the Number of Spams.

In this experiment, we created noise spams by the same way as Exp. 4 with the density $p = 0.02$. 6.10 shows the performance against the number $m$ of inserted noise spams. AM_Siz was the best when $m \leq 20$, and DCE was the best when $m > 20$. According to the increase in the number of inserted spams, the performance of DCE showed improvement while the other methods did not improve.
Exp. 6: Detecting Word Salads.

Spammers create word salads by replacing words in a skeleton document with some keywords. In this experiment, we created word salads and inserted them into the document set. The dataset used was YF4974. We created the keyword set by manually selecting from the dataset and selected a spam as the skeleton in YF8473. The keyword set consisted of 25 nouns, and the skeleton consisted of 45 words. All nouns in the skeleton were replaced in the creation of the word salads. We then created these word salads and insert them into the dataset. 6.11 shows the result. As well as Exp. 4 and 5, we define the recall by (detected word salads / inserted word salads). AM_Siz detected about 20% of word salads despite only a few word salads being inserted. However, the recall increased up to only 30%. DCE detected more word salads when the number of word salads inserted was greater than 35. The recall was approximately 100% when $m > 60$. Since Naive only detects exact copies, it could not detect word salads. AM_Len and AM_Max could not detect word salads because their threshold values were relatively higher than AM_Siz.

We also show the precision in 6.12. In this experiment, we define the precision by (number of original spams and word salads / number of reported documents). Since the threshold value of each method did not change significantly, the number of detected original spams was almost same as the experiment of Exp. 2. DCE and AM_Siz slightly improved because they could detect word salads.

6.4 Discussion

In this chapter, we studied an unsupervised spam detection algorithm. We showed that the proposed algorithm runs in linear time to the total length of the input documents. We also presented some experiment results for real and synthetic data; the real data were collected from popular bulletin boards and the synthetic data were generated
Figure 6.8: Recall and Precision curve.
Figure 6.9: Recall vs. density.
Figure 6.10: Recall vs. number of inserted noise spams.
Figure 6.11: Recall vs. number of inserted word salads.
Figure 6.12: Precision vs. number of inserted word salads.
artificially as word salad spam documents.

From the results of Exp. 6, it revealed that the performance of our method improves as the number of word salads increase, namely, as the cost of creating spams relatively goes down. Therefore we conclude that our method works successfully for such low complexity spams. As reported in [42], the ratio of splogs based on word salad to the whole splogs may be still low. However, it is always important to develop counter-measures against new and complicated spams in advance, since that spams which are hard to detect by existent methods, include query keywords, will increase rapidly on the Internet.

Very recently, Qian et al. [40] propose an unsupervised spam detection method, which has high performance to the level of state-of-the-art supervised methods. It is our future works to compare our method to such new methods [40, 23, 35].
Table 6.2: Performance of spam detection methods.

<table>
<thead>
<tr>
<th>Forum</th>
<th>Method</th>
<th>Recall</th>
<th>Precision</th>
<th>F-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>4314</td>
<td>DCE</td>
<td>0.59</td>
<td>0.95</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>AM_Len</td>
<td>0.54</td>
<td>0.92</td>
<td>0.68</td>
</tr>
<tr>
<td></td>
<td>AM_Siz</td>
<td>0.82</td>
<td>0.73</td>
<td>0.78</td>
</tr>
<tr>
<td></td>
<td>AM_Max</td>
<td>0.67</td>
<td>0.84</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>Naive</td>
<td>0.54</td>
<td>0.94</td>
<td>0.68</td>
</tr>
<tr>
<td>4974</td>
<td>DCE</td>
<td>0.63</td>
<td>0.69</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>AM_Len</td>
<td>0.4</td>
<td>0.72</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>AM_Siz</td>
<td>0.78</td>
<td>0.63</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td>AM_Max</td>
<td>0.39</td>
<td>0.70</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>Naive</td>
<td>0.52</td>
<td>0.66</td>
<td>0.58</td>
</tr>
<tr>
<td>6830</td>
<td>DCE</td>
<td>0.68</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>AM_Len</td>
<td>0.45</td>
<td>0.63</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>AM_Siz</td>
<td>0.81</td>
<td>0.48</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>AM_Max</td>
<td>0.40</td>
<td>0.71</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>Naive</td>
<td>0.54</td>
<td>0.71</td>
<td>0.62</td>
</tr>
<tr>
<td>8473</td>
<td>DCE</td>
<td>0.63</td>
<td>0.69</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td>AM_Len</td>
<td>0.54</td>
<td>0.72</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>AM_Siz</td>
<td>0.81</td>
<td>0.52</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>AM_Max</td>
<td>0.66</td>
<td>0.72</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td>Naive</td>
<td>0.54</td>
<td>0.79</td>
<td>0.64</td>
</tr>
</tbody>
</table>
Chapter 7

Training parse trees for efficient variable-to-fixed-length coding

From the viewpoint of speeding up pattern matching on compressed texts, variable-length-to-fixed-length codes (VF codes for short) are reevaluated recently [28, 24]. A VF code is a coding scheme that parses an input text into a consecutive sequence of substrings (called blocks) with a dictionary tree, which is called a parse tree, and then assigns a fixed length codeword to each substring; such codeword enables us to touch any parsed block randomly without concerning about codeword boundaries.

Several promising VF codes have been proposed so far. Maruyama et al. [33] proposed an excellent compression method, which is a variation of grammar-based compressions. They propose a \(\Sigma\)-sensitive grammar for effective grammar transform. In their practical implementation, which we call BPEX\(^1\), the method can also be viewed as a VF code since an encoded text is represented as a sequence of grammar symbols, which are represented by fixed length codewords of length 8-bits; this means the number of grammar symbols is bounded by 256. Although BPEX achieves a good compression ratio comparable to gzip, its compression speed is slow. Klein and Shapira [28] and

\(^1\)This name comes from the program implemented by Maruyama.
Kida [24] proposed independently VF codes based on suffix tree [17] (STVF code for short). In their scheme, a frequency-base-pruned suffix tree is used as a parse tree. An input text is scanned once at first to construct the parse tree, and then the text is scanned again and translated into a sequence of codewords. The compression speed of [24] is faster than that of BPEX, and the compression ratio is better than classical VF codes like Tunstall code [46], but not better than BPEX. A VF code that achieves fast compression/decompression and high compression ratio is desired.

Let Σ be an alphabet and k be the codeword bit length. Consider a text $T = t_1 t_2 \cdots t_n$ to be encoded by k-bit fixed length code, where $t_i \in \Sigma$. The aim here is to make an efficient dictionary $D$, which consists of different substrings of $T$, such that $T$ can be parsed uniquely into a sequence of entries of $D$. Each entry of $D$ is assigned a codeword of length $k$ bits, thus the number of entries in $D$ is less than or equal to $2^k$. If the text $T$ is parsed with $D$ into a sequence of $m$ blocks, $T = c_1 c_2 \cdots c_m$ ($c_i \in D$), the size of the encoded text becomes $km$ bits in addition to the size of $D$. Therefore, we want to make a dictionary such that

$$km + \sum_{c \in D} |c|$$

is minimized under $|D| \leq 2^k$. However, this problem is quite hard, as Klein and Shapira stated in [28]:

Choosing an optimal set of substrings might be intractable, since even if the strings are restricted to be the prefixes or suffixes of words in the text, the problem of finding the set is NP-complete [16], and other similar problems of devising a code have also been shown to be NP-complete in [15, 10, 26].

A natural approach is thus to suggest heuristical solutions and compare their efficiencies.

Our concern for this problem is how to construct parse trees that approximate the optimal tree better. In most VF codes, a frequency of each substring of $T$ is often used
as a clue for the approximation, since it could be related to the number of occurrences in a sequence of parsed blocks. This gives a chicken and egg problem as Klein and Shapira also stated in [28]; that is, to construct a better dictionary, which decides the partition of $T$, one has to estimate the number of entries that occurs in the partition.

In this chapter, we discuss about a method for training a parse tree of a VF code to improve its compression ratio. We propose an algorithm of reconstructing a parse tree based on the merit of each node, and we employ a heuristic approach; we apply the reconstruction many times, scanning the input text repeatedly. We can control the number of the scanning time, and also we can employ a random sampling technique to reduce the training time. We show experimentally that our method can improve VF codes comparable to gzip and BPEX with a moderate sacrifice of compression time.

The rest of this chapter is organized as follows. In Section 7.1, we discuss about VF codes, which includes brief sketches of Tunstall codes. Next, in Section 7.2, we discuss about STVF codes. Third, in Section 7.3, we introduce our method of training a parse tree. Then, in Section 7.4, we show some experimental results and describe our observations about them. Finally, we conclude in Section 7.5.

This chapter is based on the paper [53].

7.1 Variable-Length-to-Fixed-Length Codes

A VF code is a source coding that parses an input string into a consecutive sequence of variable-length substrings and then assigns a fixed length codeword to each substring. There are many variations on how they parse the input, what kind of data structures they use as a dictionary, and how they assign codewords. Among them, the method that uses a tree structure, called a parse tree, is the most fundamental and common.

Consider that we encode an input text $T \in \Sigma^*$ by a VF code of length $k$-bits codewords. Assume that a parse tree $T$ that has $\ell$ leaves is given, and each leaf in $T$
is numbered as a $k$-bits integer, where $\ell \leq 2^k$. Then, we can parse and encode $T$ with $T$ as follows:

1. Start the traversal at the root of $T$.

2. Read a symbol one by one from $T$, and traverse the parse tree $T$ by the symbol. If the traversal reaches to a leaf, then output the codeword assigned at the leaf before getting back to the root.

3. Repeat Step 2 till $T$ ends.

For example, given the text $T = AAABBACB$ and the parse tree of Fig. 7.1, the encoded sequence becomes 000/001/101/011. We call a block each factor of $T$ parsed by a parse tree. Codeword 011, for the running example, represents block $ACB$.

A decoding process of a VF code is quite simple. We can decode by replacing a codeword to a corresponding string as referring the restored parse tree.

For a memory-less information source, Tunstall code [46] is known to be an optimal VF code (see also [43]); its average code length par symbol comes asymptotically close to the entropy of the input source when the codeword length goes to infinity. It uses a parse tree called Tunstall tree, which is the optimal tree in the sense of maximizing the average block length. Tunstall tree is an ordered complete $k$-ary tree that each edge is labelled with a different symbol in $\Sigma$, where $k = |\Sigma|$. Let $\Pr(a)$ be an occurrence probability for source symbol $a \in \Sigma$. The probability of string $x_\mu \in \Sigma^+$, which is represented by the path from the root to leaf $\mu$, is $\Pr(x_\mu) = \Pi_{\eta \in \xi} \Pr(\eta)$, where $\xi$ is the label sequence on the path from the root to $\mu$ (from now on we identify a node in $T$ and a string represented by the node if no confusion occurs). Then, Tunstall tree $T^*$ can be constructed as follows:

1. Initialize $T^*$ as the ordered $k$-ary tree whose depth is 1, which consists of $k + 1$ nodes, where $k = |\Sigma|$.
2. Repeat the following while the number of leaves in $T^*$ is less than or equal to $2^k$

   (a) Select a leaf $v$ that has a maximum probability among all leaves in $T^*$.

   (b) Make $v$ be an internal node by adding $k$ children onto $v$.

Let $m$ be the number of internal nodes in $T^*$. Since the number of leaves in $T^*$ equals to $m(k - 1) + 1$, which is less than or equal to $2^k$. Hence, $m = \lfloor (2^k - 1)/(k - 1) \rfloor$.

For the other information sources, like a source with memory [45, 44], there have been proposed several coding methods that are based on Tunstall code.

Although the preprocessing time for pattern matching on a VF code depends on the size of the parse tree and the data structures for storing it, we can consider that the matching speed is almost in proportion to the compression ratio. The reason is that the time for scanning an input encoded text dominates the total time for pattern matching when the input is enough large. Therefore, the pattern matching becomes faster as the compressed data size becomes smaller; a higher compression ratio leads a smaller amount of data to be processed. Of course, the matching speed depends on what sort of algorithm we use. From the theoretical viewpoints, the VF codes we discussed above can be classified as a regular collage system[25]; thus we can obtain systematically an algorithm of Aho-Corasick type or Boyer-Moore type.

7.2 STVF Codes

A Suffix Tree based VF code (STVF code for short\textsuperscript{2}) is a coding that constructs a suitable parse tree for the input text by using a suffix tree, which is a well-known index structure that stores all substrings in the target text compactly. It is, namely, an off-line compression scheme that encodes after gathering the statistical information

\textsuperscript{2}Strictly, the methods of [24] and [28] are slightly different in detail. However, we call them the same name here since the key idea is the same.
of the whole input text beforehand. Since the suffix tree for the input text includes the text itself, we can not use the whole tree as a parse tree. We must prune it with some frequency-base heuristics to make a compact and efficient parse tree.

In the original STVF coding, codewords are assigned only to leaves in a parse tree. Some codewords are assigned to short and infrequent substrings, which cause a decline in the compression ratio. If we can assign codewords to the internal nodes, we can prune such useless leaves from the parse tree. To do this we modify the encoding procedure as follows:

1. The procedure traverses the parse tree while it can move by a symbol read from the input text.
2. If the traversal fails, then the procedure outputs the codeword of the current node without consuming the current symbol,
3. and then resumes the traversal from the root.

This encoding process is not instantaneous. Reading-ahead of just one symbol is needed. This type of VF coding is called *almost-instantaneous VF coding* (AIVF coding for short).

An AIVF coding enables us to remove infrequent edges, namely infrequent substrings, from the parse tree, and to leave only frequent edges. This flexible selection of dictionary entries contributes to an improvement in the compression ratio. We have proposed a coding method that we bring the idea of AIVF coding into STVF coding[52]. We call this variation as a STVF code hereafter instead of the original one\(^3\). We will explain the algorithm of constructing a parse tree of STVF code below.

First of all, we will make a brief sketch of suffix tree, which is the basis of the parse tree for STVF coding. For a given text \(T\), the *suffix tree* \(ST(T)\) is a compacted trie

\(^3\)This variation also employs a dynamic pruning technique stated in [51] to improve the compression speed and memory usage with a little sacrifice of the compression ratio.
that represents all the suffixes of $T$. Note that $ST(T)$ can be constructed in $O(|T|)$ time and space[55]. Formally, $ST(T)$ is defined as follows:

1. Each internal node, except the root of $ST(T)$, has at least two children.
2. Each edge is labelled by a non-empty substring of $T$.
3. For any internal node $u$, any labels of outgoing edges start with different characters each other.
4. Let the representing string $\langle v \rangle$ of a node $v$ in $ST(T)$ be the string obtained by concatenating the labels of the edges in the path from the root to $v$. Then, any substring of $T$ is a prefix of the representing string of a node in $ST(T)$.

For a node $v$ in $ST(T)$, the frequency of $v$ is defined as the number of occurrences of $\langle v \rangle$ in $T$, and denoted by $f(v)$. Since $f(v)$ can be obtained as the number of leaves in the subtree rooted at $v$, we can compute all of them in $O(|T|)$ time by a post-order traversal at once.

Next, we outline the algorithm of constructing a parse tree for a STVF code. The idea is to repeat choosing a node whose frequency is the highest in the suffix tree but not yet in the parse tree. The construction algorithm extends the parse tree on a node-by-node basis. We say that an internal node $u$ in the parse tree is complete if the parse tree contains all the children of $u$ in $ST(T)$. We do not need to assign a codeword to any complete node, since the encoding process never fail its traversals at a complete node. Figure 7.2 is an example of the parse tree constructed by the algorithm of [52] for $T = BABCABABBABCBCAB$. We can parse $T$ to five substrings with the parse tree in Fig. 7.2, as $BABC/AB/AB/BABC/BAC$, which are encoded to $101/000/000/101/110$.

For Tunstall codes and STVF codes, as we need a parse tree when we decompress an encoded text, we have to store the information for it in addition to a sequence of
codewords. For the former, all that we have to store is only the frequencies of all symbols in the alphabet, since we assume that the model of the text is a memory-less source; we can reconstruct the same tree from the frequencies. For the latter, we have to store the whole parse tree that is constructed at the encoding process. The size of the tree increases exponentially with the length of codewords. Therefore, we need to decide a suitable codeword length for compressing a text well. A practical range is about from 7 to 18 for natural language texts, DNA data, and so on. From the viewpoint of compressed pattern matching, the lengths of 8 or 16 would be the best, since we do not need any recognition of codeword boundaries and moreover we can treat the encoded text in a byte-by-byte manner.

7.3 Proposed Method

Training Parse Trees

In this section, we present a reconstruction algorithm for a readymade parse tree to improve its compression ratio. The basic idea is to exchange useless strings in the current parse tree as a result for the other strings that are expected to be frequently used. Although we must evaluate each string by some measures for doing that, it is quite hard to evaluate precisely in advance as we stated in Sec. 7. Therefore, we employ a greedy approach; we reconstruct the parse tree with two empirical measures.

We define two measures for evaluating strings. For any string $s$ in the parse tree, the accept count of $s$, denoted by $A(s)$, is defined as the number of that $s$ was used in the encoding. For any string $t$ that is not assigned a codeword, the failure count of $t$, denoted by $F(t)$, is defined as the number of that the prefix $t[1..|t| - 1]$ of $t$ was used but the codeword traversal failed at the last character of $t$. That is, $F(t)$ suggests how often $t$ likely be used if $t$ is in the parse tree. We can embed the computations of $A(s)$ and $F(t)$ in the encoding procedure. When $p = T[i..j]$ is parsed in the encoding, $A(p)$
and $F(p \cdot T[j+1])$ are incremented by one. Figure 7.3 shows an example of computing these measures.

Let $\text{pref}(s)$ be the longest proper prefix of $s$ in the parse tree. Comparing the minimum of $(|s| - |\text{pref}(s)|) \cdot A(s)$ and the maximum of $F(t)$, the reconstruction algorithm repeats to exchange $s$ and $t$ if the former is less than the latter; it removes $s$ from the parse tree and enter $t$ instead. The algorithm is as in Fig.7.4.

Note that a reconstructed parse tree is not a complete tree any longer, even if its origin is complete like Tunstall trees. Several internal nodes might be assigned codewords; thus a coding with such a tree becomes an AIVF coding.

To train a parse tree we apply the algorithm many times. For each iteration, it first encodes the input data with current parse tree. Next, it evaluates the contribution of each string in the parse tree, and then exchanges some infrequent strings for the other promising strings.

### Speeding-up by Sampling

The reconstruction of parse trees discussed above takes much time if the input text is large, since the algorithm scans the whole text many times. If we can train with small parts of the text, we can save the training time. Note here that we have to scan the whole text once to construct the initial parse tree.

Let $T$ be the input text. We consider to train with a string that consists of several pieces randomly selected from the text. Using only a part of $T$, namely a substring of $T$, does not work well even if we select randomly for each iteration, since the parse tree reconstructed by the above algorithm fits too much on the last selection. Using a set of pieces randomly selected from the whole text can work well.

Let $m$ be the number of pieces, and $B$ be the length of a piece. For given $m \geq 1$ and $B \geq 1$, we generate a sample text $S$ from $T$ at every iteration as follows:

$$S = s_1 \ldots s_m \quad (s_k = T[i_k..i_k + B - 1] \text{ for } 1 \leq k \leq m),$$
where $1 \leq i_k \leq |T| - B + 1$ is a start position of a piece that we select in a uniform random manner for each $k$. Then, $|S| = mB$. Note that the compression ratios and speeds depend on $|S|$ and $m$ in addition to the number of training iterations.

### 7.4 Experimental Results

We have implemented Tunstall coding and STVF coding with training approach that we stated in Sec. 7.3, and compared them with BPEX[33], ETDC[8], SCDC[9], gzip, and bzip2. Although ETDC/SCDC are variable-to-variable length codes, their codewords are byte-oriented and designed for compressed pattern matching. We chose 16 as the codeword lengths of both STVF coding and Tunstall coding. Our programs are written in C++ and compiled by g++ of GNU, version 3.4. We ran our experiments on an Intel Xeon (R) 3 GHz and 12 GB of RAM, running Red Hat Enterprise Linux ES Release 4.

We used DNA data, XML data, English texts, and Japanese texts to be compressed (see Table 7.1). GBHTG119 is a collection of DNA sequences from GenBank\(^4\), which is eliminated all meta data, spaces, and line feeds. DBLP2003 consists of all the data in 2003 from dblp20040213.xml\(^5\). Reuters-21578(distribution 1.0)\(^6\) is a test collection of English texts. Mainichi1991\(^7\) is from Japanese newspaper, Mainichi-Shinbun, in 1991.

**Compression ratios and speeds**

The methods we tested are the following nine: Tunstall (Tunstall codes without training), STVF (STVF codes without training), Tunstall-100 (Tunstall codes with 100

\(^5\)http://www.informatik.uni-trier.de/~ley/db/
\(^6\)http://www.daviddi Lewis.com/resources/testcollections/reuters21578/
\(^7\)http://www.nichigai.co.jp/sales/corpus.html
Table 7.1: About text files to be used.

| Texts         | size(byte) | |Σ| | Contents                                      |
|---------------|------------|---|-----|-----------------------------------------------|
| GBHTG119      | 87,173,787 | 4 | DNA sequences                              |
| DBLP2003      | 90,510,236 | 97| XML data                                    |
| Reuters-21578 | 18,805,335 | 103| English texts                               |
| Mainichi1991  | 78,911,178 | 256| Japanese texts (encoded by UTF-16)           |

times training), STVF-100 (STVF codes with 100 times training), BPEX, ETDC, SCDC, gzip, and bzip2. Figure 7.5 shows the results of compression ratios, where every compression ratio includes dictionary informations. We measured the averages of ten executions for Tunstall-100 and STVF-100.

For GBHTG119, STVF, Tunstall-100, and STVF-100 were the best in the compression ratio comparisons. Since ETDC and SCDC are word-base compression, they could not work well for the data that are hard to parse, such as DNA sequences and Unicode texts. Note that, while Tunstall had no advantage to STVF, Tunstall-100 gave almost the same performance with STVF-100. Moreover, those were between gzip and bzip2.

Figure 7.6 shows the results of compression times. STVF was much slower than Tunstall and ETDC/SCDC since it takes much time for constructing a suffix tree. As Tunstall-100 and STVF-100 took extra time for training, they were the slowest among all for any dataset.

Figure 7.7 shows the results of decompression times. Tunstall and STVF were between BPEX and ETDC/SCDC in all the data. Tunstall-100 and STVF-100 became slightly slow.
Effects of training

We examined how many times we should apply the reconstruction algorithm for sufficient training. We chose Reuter21578 as the test data in the experiments. Figure 7.8 shows the results of the effect of training for STVF and Tunstall. We can see that both compression ratios were improved rapidly as the number $k$ of iterations increases. We can also see that they seem to come close asymptotically to the same limit, which is about 32%.

We also examined how the sampling technique stated in Sec. 7.3 effects on compression ratios and speeds. We applied the sampling technique 20 times to Tunstall codes. Figure 7.9 shows the compression ratios and Figure 7.10 shows the compression speeds. We measured the average of 100 executions for each result. We observed that the compression ratio can achieve almost the same limit when the sampling size $|S|$ is 25% of the text and the number $m$ of pieces is 100. Compared with BPEX, Tunstall codes with training can overcome in compression ratios when $|S|$ is 20% and $m = 40$. The average compression time of them at that point was 30.97 seconds, while that of BPEX was 58.77 seconds.

Although STVF codes are better than Tunstall codes in compression ratios, it revealed that Tunstall codes with training are also useful from the view point of compression time.

7.5 Discussion

We presented a method for improving the compression ratio of VF codes. Marking the nodes in the parse tree that are used to the encoding, Our method replace unuseful nodes with new candidate nodes. The proposed method In this chapter, we propose a method that firstly compress the text by using the parse tree and then reconstruct the parse tree by replacing unnecessary substrings by new candidates derived by the
We showed experimentally that our method can improve the compression ratios of VF codes to the level of state-of-the-art compression methods, such as gzip and BPEX. Tunstall codes with training are about twice faster than that of BPEX in compression speed when we gain almost the same compression ratios. VF codes with training are stable and wide applicable to various data: not only English language texts, but also Unicode texts, DNA data, and so on. To compare with the variable-to-variable codes like [7] and [27], which are also designed for compressed pattern matching, is our future work. Another future work is to propose another algorithm which can reduce the trainings without sacrificing compression ratio.
Figure 7.1: An example of a parse tree.

The squares represent leaves, where codewords are assigned. The circles represent internal nodes and the numbers in the circles are their frequencies.

Figure 7.2: Parse tree of (improved) STVF coding for $T = \text{BABCABABBABCAC}$.

The squares represent the nodes assigned codewords, corresponding to the numbers in them. The circles represent the complete internal nodes.
4. Increment $A(p)$ and $F(p \cdot T[j + 1])$.

$A(p) \leftarrow A(p) + 1$

$F(p \cdot T[j + 1]) \leftarrow F(p \cdot T[j + 1]) + 1$

1. Traverse the parse tree.


3. Output codeword $C(p)$.

Figure 7.3: An example of computing accept counts and failure counts.
Algorithm ReconstructingParseTree($T, D$):

Input: A text $T = T[1..n]$ and a set $D$ of strings in the parse tree.

Output: A new set of strings.

1: $i = 1, E = \emptyset$;
2: while $i < n$
3: $p =$ the longest prefix $T[i..j]$ of $T[i..n]$ which is also included in $D$;
4: $A(p) = A(p) + 1$;
5: if $j < |T|$ then
6: $q = p \cdot T[j + 1]$;
7: $E = E \cup \{q\}$;
8: $F(q) = F(q) + 1$;
9: end if
10: $i = j + 1$;
11: end while
12: $N = \emptyset$;
13: while $D \neq \emptyset$ and $E \neq \emptyset$
14: $s =$ argmin$_{s \in D} (|s| - |\text{pref}(s)|) \cdot A(s)$;
15: $t =$ argmax$_{t \in E} F(t)$;
16: if $(|s| - |\text{pref}|) \cdot A(s) < F(t)$ then
17: $N = N \cup \{t\}$;
18: $D = D \setminus \{s\}$;
19: else
20: break;
21: end if
22: $E = E \setminus \{t\}$;
23: end while
24: return $D \cup N$;

Figure 7.4: Reconstruction algorithm for parse trees.
Figure 7.5: Compression ratios.

Figure 7.6: Compression times.
Figure 7.7: Decoding times.

Figure 7.8: The effects of training.

128
Figure 7.9: Compression ratios with sampling technique.
Figure 7.10: Compression times with sampling technique.
Chapter 8

Conclusion

8.1 Summary of the Results

In this thesis, we considered several constraints for texts and studied efficient construction of constrained suffix trees and their applications.

In Chapter 3, we proposed the word-based truncated suffix tree and its efficient construction algorithm. Given an integer $k > 0$ and a text $T$ of length $n$, we showed that the $k$-word-based suffix tree for $T$ can be constructed in $O(n)$ time, assuming that $T$ is a string over a fixed size alphabet. We extended the algorithm to store the frequency on each leaf. Experimental results showed that the word-based truncated suffix tree could reduce the space consumption from the normal suffix tree.

In Chapter 4, we studied efficient construction of property suffix trees. The construction algorithm presented by Amir et al. [1] takes $O(n \log |\Sigma| + n \log \log n)$ time for the border nodes. To improve the process, we gave an efficient algorithm for finding the border nodes and showed that the property suffix tree can be constructed in $O(n \log |\Sigma|)$ time.

In Chapter 5, we proposed the word $N$-gram tree, which is an extension of both the word-based truncated suffix tree and the word suffix tree. We also considered a
keyword extraction problem for supporting Web browsing. Using the word $N$-gram tree, we proposed a keyword extraction algorithm that runs in $O(n \log k)$ time, where $n$ is the length of the input text and $k$ is the number of extracting keywords.

In Chapter 6, we studied an unsupervised spam detection algorithm. We showed that the proposed algorithm runs in linear time to the total length of the input documents. We also presented some experiment results for real and synthetic data; the real data were collected from popular bulletin boards and the synthetic data were generated artificially as word salad spam documents. In the experiment on detecting word salads, it revealed that the performance of our method improves as the number of word salads increase, namely, as the cost of creating spams relatively goes down. Therefore we conclude that our method works successfully for such low complexity spams.

In Chapter 7, we presented a method for improving the compression ratio of VF codes. Marking the nodes in the parse tree that are used to the encoding, Our method replace unuseful nodes with new candidate nodes. The proposed method firstly compresses the text by using the parse tree and then reconstructs the parse tree by replacing unnecessary substrings by new candidates derived by the compression. We showed experimentally that our method can improve the compression ratios of VF codes to the level of state-of-the-art compression methods, such as gzip and BPEX. Tunstall codes with training are about twice faster than that of BPEX in compression speed when we gain almost the same compression ratios. VF codes with training are stable and wide applicable to various data: not only English language texts, but also Unicode texts, DNA data, and so on.

8.2 Future Researches

We proposed certain constrained suffix trees and some applications for them. However, there are many more applications in natural language processing, knowledge discovery,
and so on. Therefore it would be interesting to present another applications for the proposed constrained suffix trees.

Word-based truncated suffix tree can be considered as an extension of the character-based truncated suffix tree [36]. It is our future research to compare them in applications using natural language texts.

About the property suffix trees, Iliopoulos et al. [20] proposed another index structure that solves the property matching problem. Their structure can be constructed in $O(n)$ time. We should compare our algorithm with them. In addition, an online construction algorithm is still open.

Although we proposed the word $N$-gram tree, the construction algorithm and its complexity have not showed in this thesis. It is a future word to extend the construction algorithm of the word-based truncated suffix tree to the construction algorithm of the word $N$-gram tree.

Very recently, Qian et al. [40] propose an unsupervised spam detection method, which has high performance to the level of state-of-the-art supervised methods. It is our future works to compare our spam detection method to such new methods [40, 23, 35].

VF codes are suitable compression method for compressed pattern matching. To compare with the variable-to-variable codes like [7] and [27], which are also designed for compressed pattern matching, is our future work. Another future work is to propose another algorithm which can reduce the trainings without sacrificing compression ratio.
Bibliography


