Antidictionary Data Compression Using Dynamic Suffix Trees

by

TAKAHIRO OTA

Submitted to

The University of Electro-Communications

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Department of Information Management Science

Graduate School of Information Systems

The University of Electro-Communications

September 2007

Antidictionary Data Compression Using Dynamic Suffix Trees

Approved by Supervisory Committee:

Chairperson	:	Professor Hiroyoshi Morita
Member	:	Professor Kunikatsu Takase
Member	:	Professor Kenji Tanaka
Member	:	Professor Mamoru Hoshi
Member	:	Professor Hiroshi Nagaoka

動的な接尾辞木を用いた反辞書データ圧縮

太田　隆博

概　要

データ圧縮は，ディジタル機器における記憶容量・通信コストの削減，伝送速度の高速化を実現する手法として，情報システムの基幹技術の一つである．

圧縮手法には，大きく分けて，データを読み込みながら逐次的に圧縮を行う動的手法と，データ全体を読み込んでから圧縮を行う静的手法の二つがある．とくに，動的手法には，静的手法のようにデータを二度読みせずにすむ点や符号器で用いる符号テーブルを復号器に送らなくてすむなど，実用的に優れた特色を持つ．

これまでの動的データ圧縮手法の多くでは，入力系列を効率よく圧縮するために，辞書と呼ばれる，入力系列に現れる部分系列の集まりを表す接尾辞木というデータ構造が用いられてきた．また，接尾辞木を構築するためには高速な動的構築手法が知られている．一方，2000 年に，辞書と逆の概念である反辞書と呼ばれる入力 2 値系列に出現しない極小禁止語の集まりを利用したデータ圧縮手法 (反辞書法) が，Crochemore らにより提案された．

従来研究の結果から，辞書および反辞書は，それぞれデータ圧縮に有効なデータ構造であり，これらを組み合わせた手法は，さらによい圧縮率を与えることが期待される．本論文の主要な目的は，

- 接尾辞木による高速かつメモリ使用量を抑えた反辞書構築手法の実現
- 接尾辞木を用いた高速な動的反辞書法の実現
- 反辞書を利用した高速かつ逐次的な無ひずみ心電図データ圧縮法の実現

の 3 点である．

接尾辞木を用いて反辞書を構築する方法は従来より知られていたが，入力系列長の 2 乗に比例した計算量がかかる問題点があった．そこで，本論文では，この計算量を削減するために，接尾辞木に新しいポインタ構造を導入することによって，入力系列長に比例する計算量による反辞書構築手法を実現した．また，計算機実験により，少ないメモリ使用量で高速に反辞書構築が行えることを示した．

しかしながら，この構築法を用いて，入力系列を 1 記号読み込むたびに反辞書の更新と符号化を同時に行う動的反辞書法を実現するためには，入力系列長の 2 乗に比例した計算時間がかかる問題点があった．この計算時間を削減するために，本論文では，反辞書の構築を行わずに，接尾辞木だけを用いて反辞書による符号化が行える条件を明らかにした．さらに，この条件を用いることによって，計算時間を線形量に削減した非常に高速な動的反辞書法を実現し，さらに算術符号と組み合わせることによって圧縮効果の向上を図っている．

これまでに，2 値系列に対する算術符号の静的確率モデルとして，あらかじめ入力系列から構築した反辞書を用いる静的なデータ圧縮法 (OHY 法) が提案されているが，反辞書構築の高速化については考えられていない．また，OHY 法は，2 値系列に対しては辞書を用いた高性能な圧縮手法 (LZ 法) に匹敵する性能を持つことが示されているが，そのまま多値系列に対して適用すると復号器に送る反辞書のコストが大きくなるため，圧縮性能が悪化する問題点がある．

一方，本論文で提案している算術符号を用いた動的反辞書法は，実用的に有用な動的手法で多値系列を高速に処理できる特長を持つ．データ圧縮手法の性能比較によく用いられるデータベース (Calgary Corpus) に対して，実

験による性能評価を行った結果，本論文で提案している動的反辞書法は，従来の動的反辞書法に比べて，ほぼすべてのファイルに対して圧縮率が向上した．また，平均圧縮率による比較では，従来の動的反辞書法に比べて 3%向上し，ファイルを 2 値系列に変換して，ファイルごとに 2 つのパラメータ (分割ブロックサイズ，極小禁止語の最大長) を変化させて計算機実験で得られた OHY 法の最良の報告結果と比較しても 1%向上した．

最後に，本論文では，心電図データの無ひずみ圧縮への動的反辞書法の適用を考察している．心電図データの圧縮に関しては，測定と符号化を同時に行う動的手法で長時間のデータを少ないメモリ使用量で処理できる性能が求められる．さきに提案した動的反辞書法を心電図データ圧縮に適用すると，メモリ使用量が測定時間に比例する問題点がある．しかしながら，心電図データの大半は概周期的な波形なので，心電図全体を用いなくても，その一部分だけを用いて反辞書を構築することによって，メモリ使用量の削減と符号化処理の高速化が図れる．そこで，心電図全体に対する反辞書とほぼ同等な圧縮性能をもつためにどれくらいの長さの部分系列から反辞書を構築すればよいかについての計算式を，クーポンコレクターズ問題の適用により導出した．

心電図データ圧縮の性能比較によく用いられるデータベース (MIT-BIH Arrhythmia Database) に対して，実験による性能評価を行ったところ，提案手法は，リアルタイム伝送が可能な処理速度で，LZ 法と比較して圧縮率が 15%向上した．医学的には望まれていたが，高能率な無ひずみ圧縮が困難であった心電図データに対して，高性能でかつリアルタイム処理可能な無ひずみ圧縮手法を示した．これにより，他の生体情報などの高能率圧縮手法への応用が期待できる．

Antidictionary Data Compression Using Dynamic Suffix Trees

Takahiro Ota

Abstract

Data compression is particularly useful in systems where resources are scarce, e.g., limited bandwidth in communication systems and the capacity of storage systems. A wide variety of data compression algorithms have been proposed for inherently digital data such as text and digitized audio, image and video.

To compress an input string efficiently, data compression algorithms typically use a dictionary to construct statistical models and replace substrings with indices in the dictionary. The dictionary is the set of all substrings of an input string, and it is represented by a tree representation such as a suffix tree in many applications.

In 2000, Crochemore *et al.* [CMRS00] proposed an off-line lossless data compression algorithm using an antidictionary of an input binary string. The antidictionary is the set of all minimal strings that never appear in the string. They showed that their method, called Data Compression using Antidictionaries (DCA), achieves compression ratios that are as good as the Lempel-Ziv (LZ) algorithms [ZL77, ZL78]. In 2005, to improve the compression ratios, Ohkawa, Harada and Yamamoto [OHY05] applied an arithmetic coding to an off-line DCA method for binary strings. In other

words, the authors used antidictionaries as statistical models for an arithmetic coding. It was shown by simulation that their method, called Ohkawa-Harada-Yamamoto (OHY) method, achieves better compression ratios than the DCA method.

Both a dictionary and an antidictionary are useful for data compression. The combination of the antidictionary and the dictionary are expected to provide efficient statistical models for arithmetic coding.

The main goal of this thesis is to achieve:

- Construction of an antidictionary using a suffix tree in linear time and space.
- An on-line DCA method using dynamic suffix trees in linear time and space.
- An on-line arithmetic coding based on antidictionaries using dynamic suffix trees in linear time and space.
- One-pass lossless data compression for ElectroCardioGrams (ECGs) using antidictionaries.

Traditional construction algorithm of an antidictionary using suffix trees requires quadratic computational time with respect to a given string length. To reduce this computational time, we introduce a new kind of pointer called an MF-link in the suffix tree. By using MF-links, the proposed construction algorithm operates in linear time and space. We prove that the proposed algorithm works in linear time and space with respect to the string length, and experimental results show that the proposed algorithm is fast and memory-efficient.

On-line DCA methods can achieve better compression ratios than off-line DCA methods. However, on-line traditional DCA methods need quadratic computational time with respect to the string length, since the method requires updating the antidictionary and its encoder when a new symbol is read. To reduce this computational time, we show useful conditions to implement the on-line DCA using only suffix trees without constructing the antidictionary. By using these conditions, we propose an on-line DCA method using dynamic suffix trees without updating antidictionaries. Moreover, we apply arithmetic coding to the proposed algorithm. We prove that the proposed algorithms work in linear time and space with respect to the string length. Experimental results show that the proposed method applied with an arithmetic coding achieves better compression ratios than traditional on-line DCA for almost all files on Calgary Corpus [Cal], and this approach gives a 3% decrease in compressed file size relative the on-line DCA method and a 1% decrease in the size relative to the OHY by simulation results for the Calgary Corpus.

Finally, we propose a one-pass ECG lossless compression method using antidictionaries. The proposed algorithm constructs an encoder of the DCA from the substring of ECG, called the learning string, by means of the property that each ECG signal is an almost periodic waveform. We show that the length of learning string needed to construct the antidictionary whose size is almost the same as that of the entire string of ECG using the results of coupon collector's problem. Experimental results show that the proposed

algorithm gives 15% decrease in compressed file size relative to the LZ algorithms for ECG files of the MIT-BIH Arrhythmia Database [MIT]. Moreover, it is shown that the proposed algorithm can implement to compress the ECG files for real-time by simulation results.

Contents

List of Notations and Terminology

$\mathcal{A}(\boldsymbol{x})$	*the antidictionary of* $\boldsymbol{x}$
$\mathcal{A}_I(\boldsymbol{x})$	*the subset of* $\mathcal{A}(\boldsymbol{x})$ $\mathcal{A}_I(\boldsymbol{x}) = \{\boldsymbol{v} \mid \boldsymbol{v} = \boldsymbol{u}a \in \mathcal{A}(\boldsymbol{x}), \boldsymbol{u} \in (\mathcal{D}(\boldsymbol{x}) \backslash \mathcal{S}(\boldsymbol{x})), a \in \mathcal{X}\}$
$\mathcal{A}_L(\boldsymbol{x})$	*the subset of* $\mathcal{A}(\boldsymbol{x})$ $\mathcal{A}_L(\boldsymbol{x}) = \{\boldsymbol{v} \mid \boldsymbol{v} = \boldsymbol{u}a \in \mathcal{A}(\boldsymbol{x}), \boldsymbol{u} \in \mathcal{S}(\boldsymbol{x}), a \in \mathcal{X}\}$
$\mathcal{D}(\boldsymbol{x})$	*the dictionary of* $\boldsymbol{x}$
$\mathsf{G}(\boldsymbol{x})$	*the suffix dawg of* $\boldsymbol{x}$
$\mathsf{G}_{\mathcal{A}}(\boldsymbol{x})$	*the AD-automaton of* $\mathcal{A}(\boldsymbol{x})$
$\mathsf{G}_{\mathcal{A}_I}(\boldsymbol{x})$	*the AD-automaton of* $\mathcal{A}_I(\boldsymbol{x})$
$l(\boldsymbol{v})$	*the locus of the string* $\boldsymbol{v}$
$\mathcal{L}(p)$	*the set of all symbols between a node* p *and its children in* $\mathsf{T}(\boldsymbol{x})$ $\mathcal{L}(p) = \{a \mid \boldsymbol{w}(p)a \in \mathcal{D}(\boldsymbol{x}), a \in \mathcal{X}\}$

$\mathcal{L}_i(p)$	*the set of all symbols between a node p and its children in* $\mathsf{T}(\boldsymbol{x}^i)$ $\mathcal{L}_i(p) = \{a \vert \boldsymbol{w}(p)a \in \mathcal{D}(\boldsymbol{x}^i), a \in \mathcal{X}\}$
L_i	*the size of the set of all symbols occurring in* $\boldsymbol{x}^i$ $L_i = \vert\mathcal{L}_i(\rho)\vert$
$\mathcal{P}(\boldsymbol{x})$	*the set of all prefixes of* $\boldsymbol{x}$
$\mathcal{S}(\boldsymbol{x})$	*the set of all suffixes of* $\boldsymbol{x}$
$\boldsymbol{t}_i$	*the longest string in* $\mathcal{W}_i(\boldsymbol{x}^i)$
$\mathbb{T}(\boldsymbol{x})$	*the suffix tree of* $\boldsymbol{x}$
$\mathbb{T}^i$	*the suffix tree* $\mathbb{T}(\boldsymbol{x}^i)$
$\mathsf{T}(\boldsymbol{x})$	*the suffix trie of* $\boldsymbol{x}$
T^i	*the suffix trie* $\mathsf{T}(\boldsymbol{x}^i)$
$\mathbb{T}_{\mathcal{A}}(\boldsymbol{x})$	*the AD-tree of* $\mathcal{A}(\boldsymbol{x})$
$\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$	*the AD-tree of* $\mathcal{A}_I(\boldsymbol{x})$
$\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$	*the AD-trie of* $\mathcal{A}(\boldsymbol{x})$
$\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x})$	*the AD-trie of* $\mathcal{A}_I(\boldsymbol{x})$
$\mathsf{T}_{\mathcal{A}_L}(\boldsymbol{x})$	*the AD-trie of* $\mathcal{A}_L(\boldsymbol{x})$
$\mathsf{T}_{\mathsf{ex}}(\boldsymbol{x})$	*the extended trie of* $\boldsymbol{x}$
$\mathsf{T}_{\mathsf{I}}(x)$	$\mathsf{T}(\boldsymbol{x})$ *with all AD-nodes* ($\mathsf{T}_{\mathsf{I}}(x)$ is a subtree of $\mathsf{T}_{\mathsf{ex}}(\boldsymbol{x})$.)

$\mathbb{T}_M(\boldsymbol{x})$	$\mathbb{T}(\boldsymbol{x})$ *with all reverse MF-links*
$\mathcal{V}_i(\boldsymbol{x})$	*the subset of* $\mathcal{A}_I(\boldsymbol{x})$ $\mathcal{V}_i(\boldsymbol{x}) = \{\boldsymbol{v} \vert \boldsymbol{v} = \boldsymbol{u}a \in \mathcal{A}_I(\boldsymbol{x}), \boldsymbol{u} \in S(\boldsymbol{x}^i)), a \in \mathcal{X}\}$
$\mathcal{V}_i(\boldsymbol{x}^i)$	*the subset of* $\mathcal{A}_I(\boldsymbol{x}^i)$ $\mathcal{V}_i(\boldsymbol{x}^i) = \{\boldsymbol{v} \vert \boldsymbol{v} = \boldsymbol{u}a \in \mathcal{A}_I(\boldsymbol{x}^i), \boldsymbol{u} \in S(\boldsymbol{x}^i)), a \in \mathcal{X}\}$
$\boldsymbol{w}_i$	*the longest string in* $\mathcal{W}_i(\boldsymbol{x})$
$\boldsymbol{w}(p)$	*the path-string of a node* p
$\boldsymbol{w}(p, q)$	*the label string between a node* p *and a node* q (p is an ancestor of q.)
$\mathcal{W}_i(\boldsymbol{x})$	*the set of common substrings between proper prefixes of MFWs in* $\mathcal{A}_I(\boldsymbol{x})$ *and suffixes of* $\boldsymbol{x}^i$ $\mathcal{W}_i(\boldsymbol{x}) = \{\boldsymbol{w} \vert \boldsymbol{w}\boldsymbol{v} \in \mathcal{A}_I(\boldsymbol{x}), \boldsymbol{w} \in \mathcal{S}(\boldsymbol{x}^i), \boldsymbol{v} \in \mathcal{X}^+\} \cup \{\lambda\}$
$\mathcal{W}_i(\boldsymbol{x}^i)$	*the set of common substrings between proper prefixes of MFWs in* $\mathcal{A}_I(\boldsymbol{x}^i)$ *and suffixes of* $\boldsymbol{x}^i$ $\mathcal{W}_i(\boldsymbol{x}^i) = \{\boldsymbol{w} \vert \boldsymbol{w}\boldsymbol{v} \in \mathcal{A}_I(\boldsymbol{x}^i), \boldsymbol{w} \in \mathcal{S}(\boldsymbol{x}^i), \boldsymbol{v} \in \mathcal{X}^+\} \cup \{\lambda\}$
$\boldsymbol{x}^i$	*the prefix of* $\boldsymbol{x}$ *of length* i $\boldsymbol{x}^i = x_1x_2...x_i$
$\gamma_a(p)$	*an MF-link from a node* p *to the node* $l(a\boldsymbol{w}(p))$
λ	*the null string*
ϕ	*the empty set*

π_i	*the locus* $l(\boldsymbol{w}_i)$ *in* $\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x})$
τ_i	*the locus* $l(\boldsymbol{t}_i)$ *in* $\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x}^i)$
$\sigma(p)$	*the suffix link of a node* p
ρ	*the root node* or *the initial state*
$\mathcal{X}$	*a finite source alphabet*
$\mathcal{X}^*$	*the set of all finite strings over* $\mathcal{X}$
$\mathcal{X}^+$	$\mathcal{X}\backslash\{\lambda\}$
$\lvert\boldsymbol{x}\rvert$	*the length of a string* $\boldsymbol{x}$
$\lvert\mathcal{B}\rvert$	*the size of a set* $\mathcal{B}$
ACDCA-E	*the DCA* Encoder-AU *with an adaptive arithmetic coding for any string of* $\mathcal{X}^*$
ACDCA-D	*the DCA* Decoder-AU *with an adaptive arithmetic coding for any string of* $\mathcal{X}^*$
AD2ADT	*the construction algorithm of* $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ *from* $\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x})$
AD2D	*the conduction algorithm of* $\mathsf{T}(\boldsymbol{x})$ *from* $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$
Construct AD-Tree	*the conduction algorithm of* $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ *from* $\boldsymbol{x}$
DCA Encoder	*the encoding algorithm of the off-line DCA*
DCA Decoder	*the decoding algorithm of the off-line DCA*

DCA Encoder-AU *the encoding algorithm of the off-line DCA using* $G_{\mathcal{A}_I}(\boldsymbol{x})$

DCA Decoder-AU *the decoding algorithm of the off-line DCA using* $G_{\mathcal{A}_I}(\boldsymbol{x})$

DCA Encoder-AT *the encoding algorithm of the off-line DCA using* $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$

DCA Decoder-AT *the decoding algorithm of the off-line DCA using* $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$

ECG-DCA Encoder *the encoding algorithm of the on-line DCA for the ECG*

ECG-DCA Decoder *the decoding algorithm of the on-line DCA for the ECG*

ECG-ACDCA Encoder *the encoding algorithm of the* ECG-DCA Encoder *with an adaptive arithmetic coding*

ECG-ACDCA Decoder *the decoding algorithm of the* ECG-DCA Decoder *with an adaptive arithmetic coding*

On-line ACDCA Encoder *the encoding algorithm of the on-line arithmetic coding based on antidictionaries using dynamic suffix trees*

On-line ACDCA Decoder *the decoding algorithm of the on-line arithmetic coding based on antidictionaries using dynamic suffix trees*

On-line DCA Encoder *the encoding algorithm of the on-line DCA using dynamic suffix trees*

On-line DCA Decoder *the decoding algorithm of the on-line DCA using dynamic suffix trees*

OHY Encoder *the encoding algorithm of the OHY*

(the DCA Encoder-AU with an adaptive arithmetic coding for a binary string.)

OHY Decoder *the decoding algorithm of the OHY*

(the DCA Decoder-AU with an adaptive arithmetic coding for a binary string.)

Simple Pruning *the pruning algorithm of* $\mathcal{A}(\boldsymbol{x})$

S2AD *the construction algorithm of* $T_{\mathcal{A}}(\boldsymbol{x})$ *using* $T(\boldsymbol{x})$

S2ADB *the construction algorithm of* $T_{\mathcal{A}}(\boldsymbol{x})$ *using* $T(\boldsymbol{x})$ *for a binary string* $\boldsymbol{x}$

ST2AD *the construction algorithm of* $\mathcal{A}(\boldsymbol{x})$ *using* $\mathbb{T}(\boldsymbol{x})$

ST2ADT *the construction algorithm of* $T_{\mathcal{A}}(\boldsymbol{x})$ *using* $\mathbb{T}(\boldsymbol{x})$

Compression ratio *an indicator to evaluate compression performance*

$$\text{Compression ratio} = \frac{\text{compressed file size}}{\text{original file size}}$$

Chapter 1

Introduction

Data compression is a useful technique to conserve resources, e.g., bandwidth in communication systems or space in storage systems. Various data compression methods have been proposed for the transmission and storage of digital data such as text, audio, image and video.

Data compression methods can be categorized into two classes: *lossless* and *lossy* methods. Lossless data compression allows the original data to be reconstructed exactly from the compressed data. This is appropriate for applications that require the decompressed data to be identical to the original data such as a text, executable files and images. On the other hand, lossy data compression methods reconstruct approximate the original data from the compressed data. Such methods are commonly used to compress multimedia data such as audio, image and video.

Lossless data compression method can be classified into two classes, one for *statistical* methods and *dictionary* methods. A statistical compression method assigns a symbol to a code by means of a statistical model and an

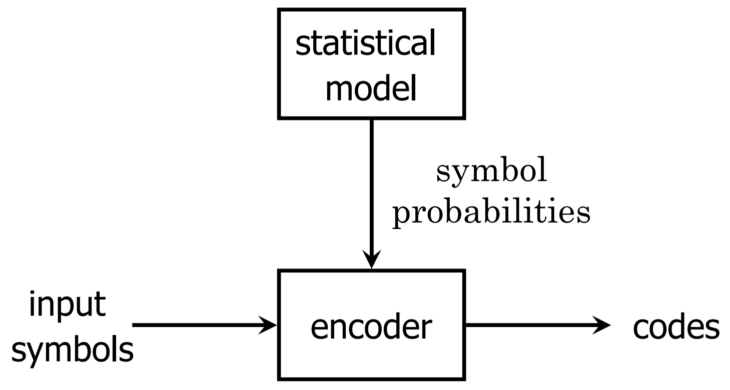


Figure 1.1: A general statistical data compression scheme.

encoder. The code for a symbol depends on the probability that the symbol occurs in an input string. Figure 1.1 shows a general statistical data compression scheme. A statistical model provides the probability distribution for each symbol. Practical statistical compression methods generate a statistical model and use entropy coding such as Huffman coding [Huf52] or arithmetic coding [Ris76, Pas76, WC87] to assign a symbol or a word to the code.

An *off-line*, that is *static*, compression scheme constructs the statistical model from an input string in the first pass, and encodes the input string by using the statistical model in the second pass. Off-line compression schemes require at least two passes of an input string, and they need to send the fixed model to the decoder. On the other hand, an *on-line* scheme requires only a single pass over an input string and sends no model to the decoder. An on-line scheme is also called a *dynamic*, *adaptive* or *universal* scheme. The encoder needs a modification of the model whenever a new symbol is read, and in the decoding process, the identical modification is carried out in order for the decoder to reproduce the identical model. The practical methods of constructing statistical models such as *tree* structures have been proposed [Ris83, CW84, Mof90, CTW95, WST95].

The dictionary is defined as the set of all substrings occurring in an input string. Dictionary compression methods use the dictionary to compress an input string. This method generates the dictionary for an input string, and compression is achieved by searching the dictionary for fragments of the input string and replacing them with an index into the dictionary. The practical two methods proposed by Lempel and Ziv called LZ77 and LZ78 [ZL77, ZL78] and their variants [SS82, Wel84, FG89, MK92] are well-known as dictionary data compression methods.

Dictionaries are not only useful for dictionary compression methods, but also for statistical compression methods. A tree model such as a *suffix trie* or a *suffix tree* is used to represent the dictionary for a given string. A suffix tree is a compacted representation of a suffix trie. The tree models store all substrings occurring in a given string, and linear algorithms to construct the suffix tree for a given string are proposed [Wei73, McC76, Ukk95, Far97]. Larsson proposed a statistical compression method using suffix trees [Lar96]. A directed acyclic word graph (dawg) is also used to represent the dictionary. The minimized representation of the suffix trie is known as the *suffix dawg* [BBHECS85, Cro86].

Traditional applications of data compression have gathered elements of an input string such as the dictionary and used them to compress an input string.

In 2000, Crochemore *et al.* proposed an off-line lossless data compression method by means of an *antidictionary* in an input binary string [CMRS00].

The antidictionary is the set of all minimal strings, called *Minimal Forbidden Words (MFWs)*, that never appear in an input string. Their method, called Data Compression using Antidictionaries (DCA), was applied to files of an well-known database for data compression called the Calgary Corpus [Cal] and it was shown that the DCA method achieves almost the same compression ratios as the Lempel-Ziv algorithm. They proved that the DCA method attains the entropy for balanced binary sources which are unifilar Markov sources [Ash90] with two output symbols of equal probabilities or of probability one and zero. Since the DCA method works with an off-line manner, it must send an antidictionary to its decoder. Morita and the author proved the size of the antidictionary of a binary sting is smaller than or equal to that of its dictionary [MO05]. They also proposed an algorithm to reconstruct its dictionary from a given antidictionary. Furthermore, it was shown that the upper bound on the size of an antidictionary is given by linear order [CEGM04, MO05]. Fayolle showed, in his doctoral thesis [Fay06], for a binary string of length n generated by a memoryless binary source, the average number of elements of the antidictionary. Its size is asymptotically $Kn/h + o(n)$ where h is the entropy of the model and K is a constant explicitly computed.

Figure 1.2 shows schemes of the DCA. In the DCA method, an antidictionary is represented by a tree structure called *AD-trie* to reduce memory space. The AD-trie is constructed by using a suffix dawg in linear time, while the construction algorithms of the AD-trie using a suffix tree or a suffix trie requires a quadratic computational time. The DCA method uses a deterministic finite automaton called *AD-automaton* as its encoder. By using the

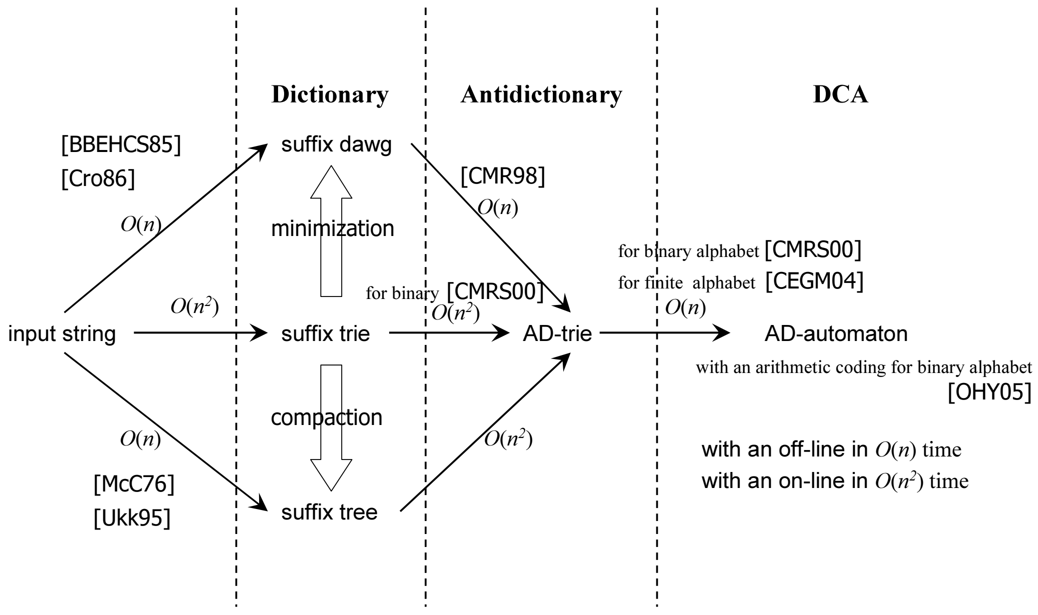


Figure 1.2: Schemes of the DCA.

AD-automaton, the DCA method can encode a given string in linear time. In 2005, to improve the compression ratios of the DCA method, Ohkawa *et al.* used the AD-automaton of the antidictionary of a binary string as a statistical model for an arithmetic coding [OHY05]. It was shown that their method, called Ohkawa-Harada-Yamamoto (OHY) method, achieved better compression ratios for files of the Calgary Corpus than the DCA methods by computer simulation results.

1.1 Objectives

Both a dictionary and an antidictionary are useful for constructing a statistical model, and a tree representation of the dictionary such as a suffix trie or a suffix tree is also useful for a statistical model. In this thesis, we propose a tree model based on a dictionary and an antidictionary and apply the model to an arithmetic coding.

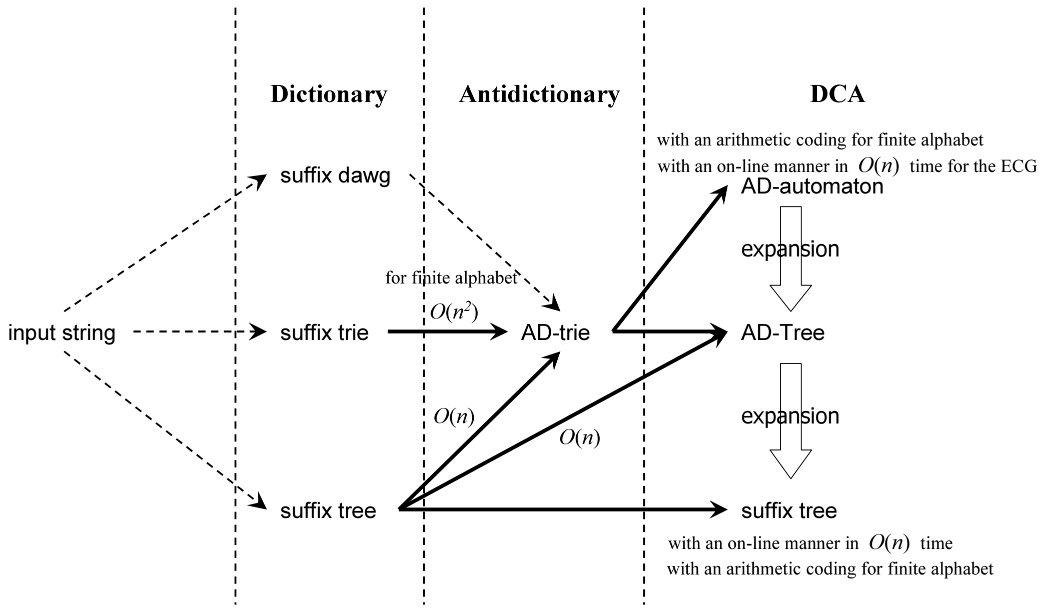
The main goal of this thesis is to achieve:

- construction of the antidictionary using a suffix tree in linear time and space.
- an on-line DCA method using the tree model in linear time and space.
- an on-line statistical data compression using the tree model based on a dictionary and an antidictionary in linear time and space.

As shown in Figure 1.2, the construction algorithm of an antidictionary using a tree structure requires quadratic computational time with respect to a string length. To construct the antidictionary using the suffix tree in linear time, we will introduce a new kind of pointers, called *MF-links*, to efficiently traverse through the suffix tree. The MF-links are as essential as the suffix links for the construction of an antidictionary in linear time.

Then, we propose a new tree structure, called *AD-tree*, which is the subtree of the suffix tree with reverse MF-links. The DCA algorithm using an AD-tree can obtain the same output as that of the DCA using the AD-automaton in linear time and space with an off-line manner. The properties of the AD-tree are useful for producing an on-line linear DCA algorithm. By using those properties, we propose an on-line DCA using dynamic suffix trees. Our algorithm works in linear time, while the traditional on-line DCA algorithms require quadratic computational time.

Figure 1.3 shows our methods for the DCA. In Figure 1.3, the solid lines represent our methods in this thesis.



Dictionary
Antidictionary
DCA
with an arithmetic coding for finite alphabet
with an on-line manner in $O(n)$ time for the ECG
suffix dawg
AD-automaton
expansion
for finite alphabet
$O(n^2)$
input string
suffix trie
AD-trie
AD-Tree
expansion
$O(n)$
$O(n)$
suffix tree
suffix tree
with an on-line manner in $O(n)$ time
with an arithmetic coding for finite alphabet

Figure 1.3: Our methods for the DCA.

1.2 Organization of the Thesis

This thesis is organized as follows.

Chapter 2 details the basic definitions, representations of a dictionary such as a *suffix trie*, a *suffix tree* and a *suffix dawg*, and tree representations of an antidictionary such as an *extended trie* and an *AD-trie*. The review of construction of AD-tries is also detailed. Then, we prove the upper bound on the number of nodes of an AD-trie. Furthermore, we generalize the construction algorithm of an antidictionary for binary strings using a suffix trie to any string over finite alphabet.

Chapter 3 details the review of schemes of the DCA, and an adaptive arithmetic coding based on the antidictionary of a binary string using the AD-automaton, that is the OHY algorithm. Then, we generalize the OHY algorithm to any string over finite alphabet.

Chapter 4 gives a construction algorithm of an antidictionary using a suffix tree, and proves that computational complexity of the proposed algorithm is linear time and space. We show that its effectiveness by simulation results. Then, we propose a linear construction algorithm of an AD-trie using a suffix tree.

Chapter 5 gives two new schemes of the DCA method by means of tree structures such as an *AD-tree* and a suffix tree. The DCA method using the AD-tree works with an off-line in linear time and space, and the properties of the AD-tree are useful for producing an on-line DCA method with linear complexity. By using those properties, we propose an on-line DCA method using dynamic suffix trees, and prove that the proposed algorithm works with linear complexity. Moreover, we propose an on-line arithmetic coding based on antidictionaries using dynamic suffix trees. Experimental results show its effectiveness.

Chapter 6 gives a new on-line DCA method for Electrocardiogram (ECG). The proposed algorithm constructs an AD-automaton from the substring of an input string, and we study on the length of the substring needed to construct the antidictionary whose size is almost same as that of the entire string of ECG using the results of coupon collector's problems. We show its effectiveness by simulation results.

Chapter 7 summarizes this thesis.

Chapter 2

Basic Definitions and Data Structures

2.1 Alphabet and Strings

Let $\mathcal{X}$ be a finite source *alphabet* $\{1, 2, \ldots, m\}$. A *string* is a sequence of symbols that are taken from the alphabet $\mathcal{X}$. The length of a string $\boldsymbol{x}$ is denoted by $|\boldsymbol{x}|$. Let $\mathcal{X}^*$ be the set of all finite strings over $\mathcal{X}$, including the null string of length zero, denoted by λ. Let $\mathcal{X}^+$ be the set $\mathcal{X}^* \backslash \{\lambda\}$.

If string $\boldsymbol{x} = \boldsymbol{y}\boldsymbol{z}$, then $\boldsymbol{y}$ is a *prefix* of $\boldsymbol{x}$ and $\boldsymbol{z}$ is a *suffix* of $\boldsymbol{x}$ where $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathcal{X}^*$. String $\boldsymbol{x}$ is not only a prefix of $\boldsymbol{x}$ but also a suffix of $\boldsymbol{x}$. Let $\boldsymbol{x}^k$ be the prefix of length k of $\boldsymbol{x}$, and we define that $\boldsymbol{x}^0 = \lambda$. A prefix $\boldsymbol{y}$ or suffix $\boldsymbol{z}$ of $\boldsymbol{x}$ is called *proper* if $\boldsymbol{x} = \boldsymbol{y}\boldsymbol{w}\boldsymbol{z}$ where $\boldsymbol{w} \in \mathcal{X}^+$. A *substring* of $\boldsymbol{x}$ is a prefix of a suffix of $\boldsymbol{x}$.

For a string $\boldsymbol{x} = x_1 x_2 \ldots x_n \in \mathcal{X}^*$ of length n, let $\mathcal{P}(\boldsymbol{x})$ be the set of all prefixes of $\boldsymbol{x}$, that is,

$$\mathcal{P}(\boldsymbol{x}) = \{x_1 \dots x_i | 1 \leq i \leq n\} \cup \{\lambda\}, \tag{2.1}$$

and we define that $\mathcal{P}(\lambda) = \{\lambda\}$.

Similarly, let $\mathcal{S}(\boldsymbol{x})$ be the set of all suffixes of $\boldsymbol{x}$, that is,

$$\mathcal{S}(\boldsymbol{x}) = \{x_i \dots x_n | 1 \leq i \leq n\} \cup \{\lambda\}, \tag{2.2}$$

and we define that $\mathcal{S}(\lambda) = \{\lambda\}$.

2.2 Dictionaries and Antidictionaries

The *dictionary* $\mathcal{D}(\boldsymbol{x})$ is defined as the set of all substrings of $\boldsymbol{x}$, that is,

$$\mathcal{D}(\boldsymbol{x}) = \{x_i x_{i+1} \dots x_j | 1 \leq i \leq j \leq n\} \cup \{\lambda\}. \tag{2.3}$$

A string $\boldsymbol{v} = v_1 v_2 \dots v_k$ of length $k \geq 2$ with the properties

$$i)\ \boldsymbol{v} \in \mathcal{X}^* \backslash \mathcal{D}(\boldsymbol{x}) \tag{2.4}$$

$$ii)\ v_1 v_2 \dots v_{k-1} \in \mathcal{D}(\boldsymbol{x}) \tag{2.5}$$

$$iii)\ v_2 v_3 \dots v_k \in \mathcal{D}(\boldsymbol{x}) \tag{2.6}$$

is called a *Minimal Forbidden Word (MFW)* of $\boldsymbol{x}$. The *antidictionary* of $\boldsymbol{x}$, denoted by $\mathcal{A}(\boldsymbol{x})$, is defined as the set of all MFWs of $\boldsymbol{x}$. For example, the dictionary $\mathcal{D}(\boldsymbol{x})$ and the antidictionary $\mathcal{A}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$ are

$$\begin{aligned}\{\lambda, 1, 2, 3, 12, 21, 22, 23, 31, 122, 123, 212, 221, 231, 1221, 1231, 2123, 2212,\\ 12212, 21231, 22123, 122123, 221231, 1221231\}\end{aligned} \tag{2.7}$$

and

$$\{11, 13, 32, 33, 121, 222, 223, 312, 2122\}, \tag{2.8}$$

respectively. It can be verified that the string 121 is an MFW, because it is not included in $\mathcal{D}(\boldsymbol{x})$ and it fulfills (2.4), (2.5) and (2.6).

We use the function $|\cdot|$ to represent not only the length of a string $\boldsymbol{x}$ but also the cardinality or size of a set $\mathcal{B}$. The difference between $|\boldsymbol{x}|$ and $|\mathcal{B}|$ is clear from a context. The empty set is denoted by ϕ. Let $|\mathcal{B}|$ be zero in case of $\mathcal{B} = \phi$. For example, $|\mathcal{A}(\boldsymbol{x})|$ for $\boldsymbol{x} = 1221231$ in (2.8) is 9.

2.3 Tree and Automaton Representations of the Dictionary

2.3.1 Suffix Tries

We first introduce basic tree terminology. A *tree* is a structure defined recursively to be either a single *external* node or an *internal* node that is connected to at least one tree. The nodes directly below a node are called its *children*; nodes farther down are called *descendants*; the node directly above a node is called its *parent*; nodes farther up are called *ancestors*. Each node has exactly one parent node, except the node at the top of a tree called the *root* denoted by ρ. The root ρ has no parent. External nodes, called *leaves*, have no child. An *edge* is defined as a link between a node and its child. A *path* in the tree is defined by a sequence of connected nodes.

The *suffix trie* $\mathcal{T}(\boldsymbol{x})$ is a tree structure that stores each suffix of $\boldsymbol{x}$ as a path from ρ to a node in $\mathcal{T}(\boldsymbol{x})$. Every edge in $\mathcal{T}(\boldsymbol{x})$ is labeled with a symbol in $\mathcal{X}$, and every path from ρ to a leaf corresponds to a suffix of $\boldsymbol{x}$. Any string in $\mathcal{D}(\boldsymbol{x})$ can be represented as a path from ρ to a node in $\mathcal{T}(\boldsymbol{x})$.

The string associated with the path from ρ to a node p is called the *path-string* and is denoted by $\boldsymbol{w}(p)$. Let $\boldsymbol{w}(\rho)$ be the null string λ.

On the other hand, the node such that $\boldsymbol{v} = \boldsymbol{w}(p)$ is called the *locus* of $\boldsymbol{v}$ and the node p is denoted by $l(\boldsymbol{v})$. If a string $\boldsymbol{v} \in \mathcal{D}(\boldsymbol{x})$, then there exists the node p of $\mathcal{T}(\boldsymbol{x})$ such that $p = l(\boldsymbol{v})$.

For any node p in $\mathcal{T}(\boldsymbol{x})$, let $\mathcal{L}(p)$ be the set of all symbols that are associated with all edges sprouting from p in $\mathcal{T}(\boldsymbol{x})$, that is,

$$\mathcal{L}(p) = \{a | \boldsymbol{w}(p)a \in \mathcal{D}(\boldsymbol{x}), a \in \mathcal{X}\}. \tag{2.9}$$

The set $\mathcal{L}(\rho)$ denotes the set of all symbols occurring in $\boldsymbol{x}$.

An internal node $p \neq \rho$ in $\mathcal{T}(\boldsymbol{x})$ that has a single child is called *implicit*, while an internal node with at least two children, a leaf and ρ are called *explicit*.

Figure 2.1 shows $\mathcal{T}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$. It consists of eighteen internal nodes (ρ, $p_1, ..., p_{17}$), six leaves ($q_1, q_2, q_3, q_4, q_5, q_6$), and twenty-three edges.

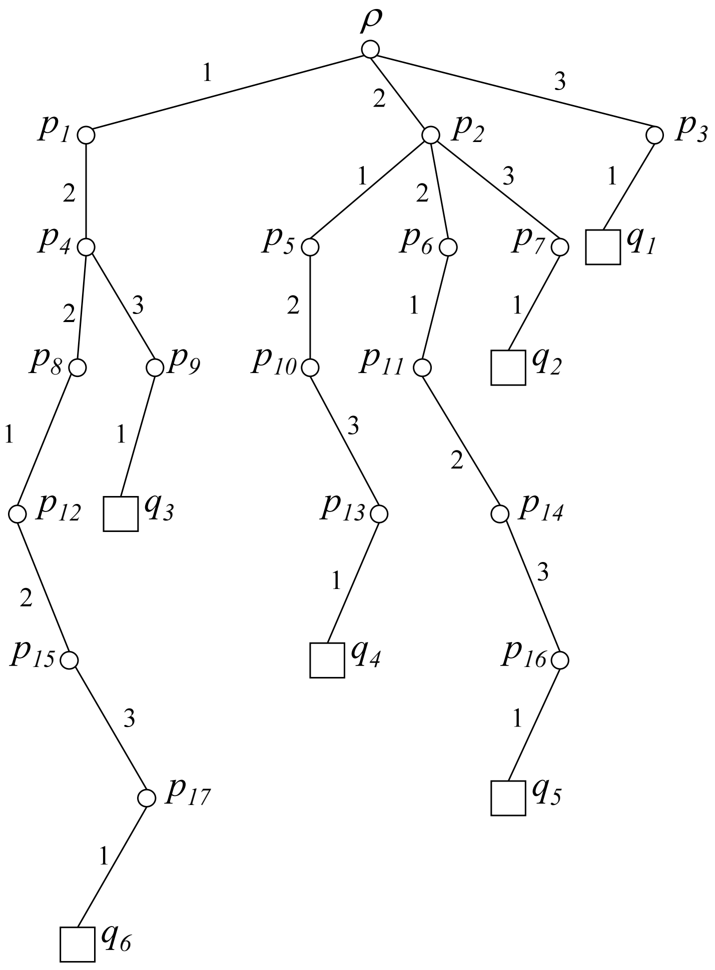


Figure 2.1: Suffix trie $T(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

In Figure 2.1, for example, the set $\mathcal{L}(\rho) = \{1, 2, 3\}$, $\mathcal{L}(p_4) = \{2, 3\}$ and $\mathcal{L}(p_6) = \{1\}$.

The trie $T(\boldsymbol{x}^i)$ denotes the suffix trie of $\boldsymbol{x}^i$. We use the simple notation such as T^i instead of $T(\boldsymbol{x}^i)$. Note that T^n is $T(\boldsymbol{x})$ of the string $\boldsymbol{x}$ of length n. For any node p in T^i, we define $\mathcal{L}_i(p)$ as the following Eq. (2.10). It is similar to $\mathcal{L}(p)$ in (2.9).

$$\mathcal{L}_i(p) = \{a | \boldsymbol{w}(p)a \in \mathcal{D}(\boldsymbol{x}^i), a \in \mathcal{X}\}. \tag{2.10}$$

The set $\mathcal{L}_i(\rho)$ denotes the set of all symbols occurring in $\boldsymbol{x}^i$. Let L_i be the size $|\mathcal{L}_i(\rho)|$. Therefore, L_n is the size of the set of all symbols occurring in $\boldsymbol{x}$ of length n. These notations are useful to represent suffix tries constructed with an *on-line* manner.

The number of nodes in $\mathcal{T}(\boldsymbol{x})$ corresponds to the size of dictionary $|\mathcal{D}(\boldsymbol{x})|$. Janson *et al.* [JLS04] investigated the average size of $\mathcal{D}(\boldsymbol{x})$ of a random string $\boldsymbol{x}$ of length n that was generated by a mixing model. It was shown that, asymptotically, its size is equal to $n^2/2$. This implies that the average number of nodes of $\mathcal{T}(\boldsymbol{x})$ is of order $O(n^2)$. It takes $O(n^2)$ time to construct $\mathcal{T}(\boldsymbol{x})$ for the string $\boldsymbol{x}$ of length n since all nodes of $\mathcal{T}(\boldsymbol{x})$ are created by a construction algorithm.

2.3.2 Suffix Trees

The *suffix tree* $\mathbb{T}(\boldsymbol{x})$ is a compacted representation of $\mathcal{T}(\boldsymbol{x})$. The tree $\mathbb{T}(\boldsymbol{x})$ is obtained by reducing the number of nodes in $\mathcal{T}(\boldsymbol{x})$. The number of nodes in $\mathcal{T}(\boldsymbol{x})$ can be reduced by directly connecting the explicit nodes and relabeling the edges, thus eliminating all implicit nodes. The resulting tree $\mathbb{T}(\boldsymbol{x})$ has the same topology as the trie $\mathcal{T}(\boldsymbol{x})$. Every edge of $\mathbb{T}(\boldsymbol{x})$ from an (explicit) node p to an (explicit) node r is labeled with a *label string* $\boldsymbol{w}(p, r)$ that corresponds to the string associated with the path from p to r in the corresponding $\mathcal{T}(\boldsymbol{x})$. It follows that $\boldsymbol{w}(r) = \boldsymbol{w}(p)\boldsymbol{w}(p, r)$. Since $\boldsymbol{w}(p, r)$ is a substring of $\boldsymbol{x}$, that is,

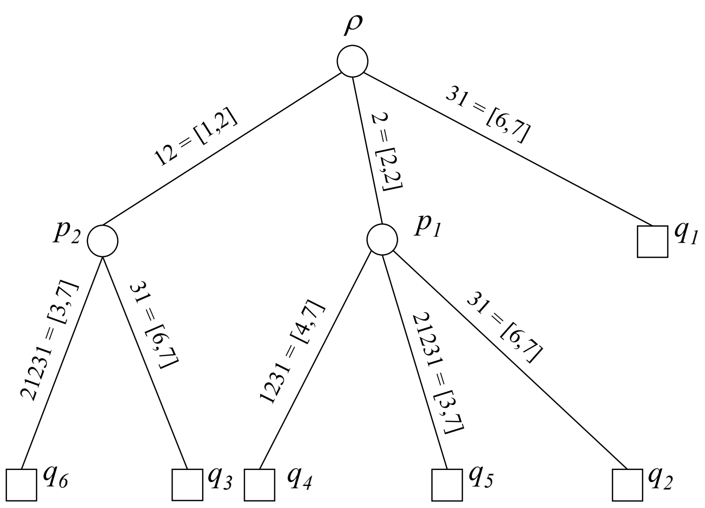


Figure 2.2: Suffix tree $\mathbb{T}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

$\boldsymbol{w}(p, r) = x_i x_{i+1} \dots x_j$, the label string can be represented by the pair $[i, j]$.

Figure 2.2 shows $\mathbb{T}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$. It consists of three internal nodes (ρ, p_1, p_2), six leaves ($q_1, q_2, q_3, q_4, q_5, q_6$), and eight edges. For example, the path-strings $\boldsymbol{w}(q_6) = 1221231$ and $\boldsymbol{w}(q_3) = 1231$ share the common prefix $\boldsymbol{w}(p_2) = 12$, and the label string $\boldsymbol{w}(p_2, q_6) = 21231$ corresponds to a suffix of $\boldsymbol{w}(q_6) = 1221231$. Using the aforementioned shorthand notation, the label string 21231 is represented by $[3, 7]$.

To make implicit nodes more easily understandable in these figures, we use another representation of $\mathbb{T}(\boldsymbol{x})$ as shown in Figure 2.3 where all the implicit nodes are virtually indicated by small circles. In this figure, small circles, large circles and squares represent implicit nodes, explicit nodes and leaves, respectively. In $\mathbb{T}(\boldsymbol{x})$, an implicit node q is represented by $(p, [i, j])$ where p is an explicit node that is the nearest ancestor of q and $[i, j]$ is the shorthand notation of the label string $\boldsymbol{w}(p, q)$. For example, implicit node

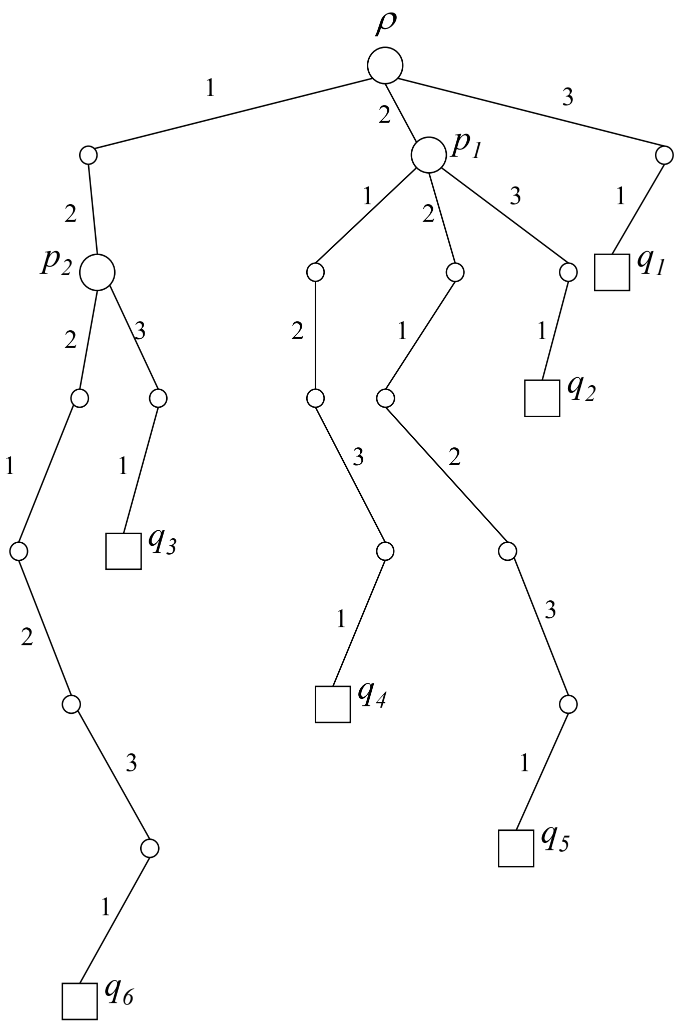


Figure 2.3: Suffix tree $\mathbb{T}(\boldsymbol{x})$ with implicit nodes for $\boldsymbol{x} = 1221231$.

p_{13} of $\mathcal{T}(\boldsymbol{x})$ in Figure 2.1 is represented by $(p_1, [4, 6])$ in Figure 2.3, where $[4, 6]$ is label string 123.

For any internal node $p \neq \rho$ in $\mathbb{T}(\boldsymbol{x})$ or $\mathcal{T}(\boldsymbol{x})$, we can write $\boldsymbol{w}(p) = a\boldsymbol{v}$, where $a \in \mathcal{X}$ and $\boldsymbol{v} \in \mathcal{X}^*$. Let q be a node such that $\boldsymbol{w}(q) = \boldsymbol{v}$, and we establish a pointer from p to q, called a *suffix link*, and denoted by $\sigma(p)$. Suffix links play a key roll in linear complexity algorithms for the construction of

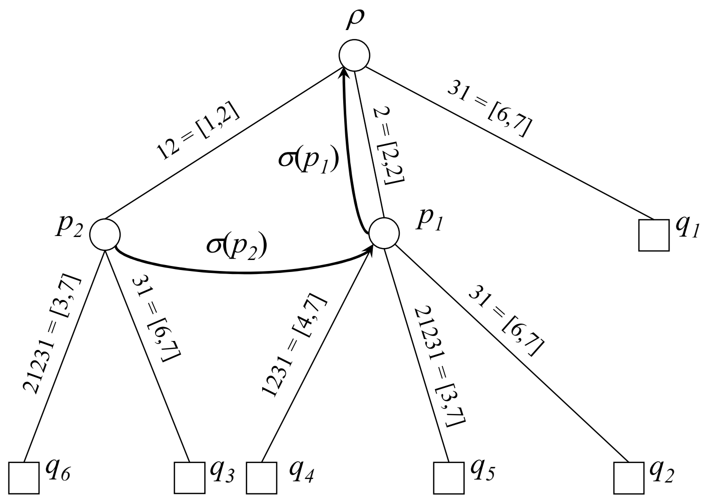


Figure 2.4: Suffix tree $\mathbb{T}(\boldsymbol{x})$ with suffix links for $\boldsymbol{x} = 1221231$.

suffix trees [Gus97]. Figure 2.4 shows $\mathbb{T}(\boldsymbol{x})$ with suffix links. A curved line denotes a suffix link for $\boldsymbol{x} = 1221231$. As shown in Figure 2.4, the suffix link $\sigma(p_2)$ and $\sigma(p_1)$ are p_1 and ρ, respectively.

If a node p in $\mathcal{T}(\boldsymbol{x})$ has a child q and the edge from p to q is labeled by symbol a, then q is alternatively represented by (p, a). The same notation is useful to represent a child in $\mathbb{T}(\boldsymbol{x})$. For an internal node p of $\mathbb{T}(\boldsymbol{x})$, there is a child q of p in $\mathbb{T}(\boldsymbol{x})$, and q is also written as (p, a) if $\boldsymbol{w}(q) = \boldsymbol{w}(p)a\boldsymbol{v}$ for $\boldsymbol{v} \in \mathcal{X}^*$ since, in $\mathbb{T}(\boldsymbol{x})$, $\boldsymbol{w}(p, q)$ can be identified by only its first symbol. In Figure 2.1 and Figure 2.2, for example, the node $(p_4, 2)$ in $\mathcal{T}(\boldsymbol{x})$ is p_8 and $(p_2, 2)$ in $\mathbb{T}(\boldsymbol{x})$ is q_6.

The tree $\mathbb{T}(\boldsymbol{x}^i)$ denotes the suffix tree of $\boldsymbol{x}^i$. Similarly the trie $\mathcal{T}^i$, we use the simple notation such as $\mathbb{T}^i$ instead of $\mathbb{T}(\boldsymbol{x}^i)$.

The number of nodes in $\mathbb{T}(\boldsymbol{x})$ is linear order with respect to the string length n, while the number of nodes in $\mathcal{T}(\boldsymbol{x})$ is quadratic order. McCreight showed $\mathbb{T}(\boldsymbol{x})$ has at most $2n$ nodes [McC76].

The straightforward algorithm to construct $\mathbb{T}(\boldsymbol{x})$ requires quadratic time in the worst case as well as $\mathsf{T}(\boldsymbol{x})$ for a string $\boldsymbol{x}$ of length n. Apostolico *et al.* [AS92] showed that its time complexity is $O(n \log n)$ in the expected case.

Four construction algorithms for suffix trees with linear time are well known [Wei73, McC76, Ukk95, Far97]. In 1973, the first algorithm was proposed by Weiner [Wei73]. Although it is a new and innovative algorithm, it is not common use in practice because the others are more efficient with respect to space. The second algorithm was presented by McCreight in 1976 [McC76]. It has a space saving improvement over Weiner's algorithm. While the above two algorithms work with an off-line manner, the third algorithm proposed by Ukkonen [Ukk95] in 1995 has important property such as an on-line manner. It is useful for applications in sequential data processing such as the string matching problem, data compression and so on. In 1997, the fourth algorithm proposed by Farach [Far97] works in a linear time (independent of the alphabet size) for integer alphabets.

2.3.3 Suffix Dawgs

A *suffix directed acyclic word graph (suffix dawg)* $\mathsf{G}(\boldsymbol{x})$ is a data structure that stores all suffixes of a string $\boldsymbol{x}$ [BBHECS85]. It is represented as a finite deterministic automaton of which every vertex is called the *state* of the dawg and every edge labeled with a symbol in $\mathcal{X}$ is the possible transition into a next state. The dawg $\mathsf{G}(\boldsymbol{x})$ is similar to $\mathsf{T}(\boldsymbol{x})$ and $\mathsf{G}(\boldsymbol{x})$ is more memory-efficient than $\mathsf{T}(\boldsymbol{x})$. To reduce memory space, $\mathsf{G}(\boldsymbol{x})$ eliminates a prefix and a suffix redundancy by sharing common prefixes and suffixes among substrings, while $\mathsf{T}(\boldsymbol{x})$ reduces a prefix redundancy by sharing common prefixes. In Theorem 6.1 [CR02], it is proved that $\mathsf{G}(\boldsymbol{x})$ has less than $2n$ states with respect to a string length n.

Figure 2.5 depicts $\mathsf{G}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$, where ρ is the *initial state*, and the *end-state* q_1 is represented by a square.

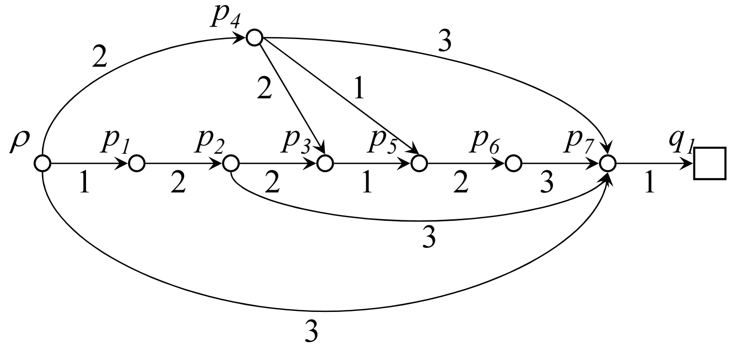


Figure 2.5: Suffix dawg $\mathsf{G}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

If a state p in $\mathsf{G}(\boldsymbol{x})$ has a next state q and the edge from p to q is labeled by symbol a, then q is alternatively represented by (p, a). We use the same notation for $\mathsf{T}(\boldsymbol{x})$ and $\mathbb{T}(\boldsymbol{x})$. Linear algorithms to construct a suffix dawg with an off-line and an on-line manner are proposed [BBHECS85, Cro86](cf. [CR02]).

2.4 Tree Representations of the Antidictionary

2.4.1 Extended Tries

The *extended trie* $\mathcal{T}_{ex}(\boldsymbol{x})$ is constructed by adding new nodes to any node q of $\mathcal{T}(\boldsymbol{x})$ as follows: If $c \in \mathcal{L}(\rho)$ and $c \notin \mathcal{L}(q)$ are satisfied, then a new node r is connected to q and $\boldsymbol{w}(q, r)$ is labeled by c. Every internal node of $\mathcal{T}_{ex}(\boldsymbol{x})$ has L_n children. In other words, $\mathcal{T}_{ex}(\boldsymbol{x})$ is L_n*-ary* tree [Gus97]. Figure 2.6 depicts $\mathcal{T}_{ex}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$, where the newly created nodes and edges are represented by triangles and dashed lines, respectively. A triangle node is a leaf of $\mathcal{T}_{ex}(\boldsymbol{x})$.

Morita *et al.* [MO05] proved the following Theorem with respect to a necessary and sufficient condition on MFWs.

Theorem 1 (Morita and Ota, 2005). *For a leaf p in $\mathcal{T}_{ex}(\boldsymbol{x})$, the path-string $\boldsymbol{w}(p)$ is an MFW of $\boldsymbol{x}$ if and only if $\sigma(p)$ is an internal node in $\mathcal{T}_{ex}(\boldsymbol{x})$.*

Proof. This proof is omitted here. (see [MO05]) □

Straightforward algorithms for the construction of $\mathcal{A}(\boldsymbol{x})$ need to examine each path-string of all leaves of $\mathcal{T}_{ex}(\boldsymbol{x})$ whether Eq. (2.6) is satisfied. It follows that the computational time for the straightforward construction of $\mathcal{A}(\boldsymbol{x})$ is of order $O(n^2)$.

2.4.2 AD-Tries

The *AD-trie* $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ of a string $\boldsymbol{x}$ is a tree structure that stores each MFW of $\mathcal{A}(\boldsymbol{x})$ as a path from ρ to a leaf in $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$. Every edge in $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ is labeled

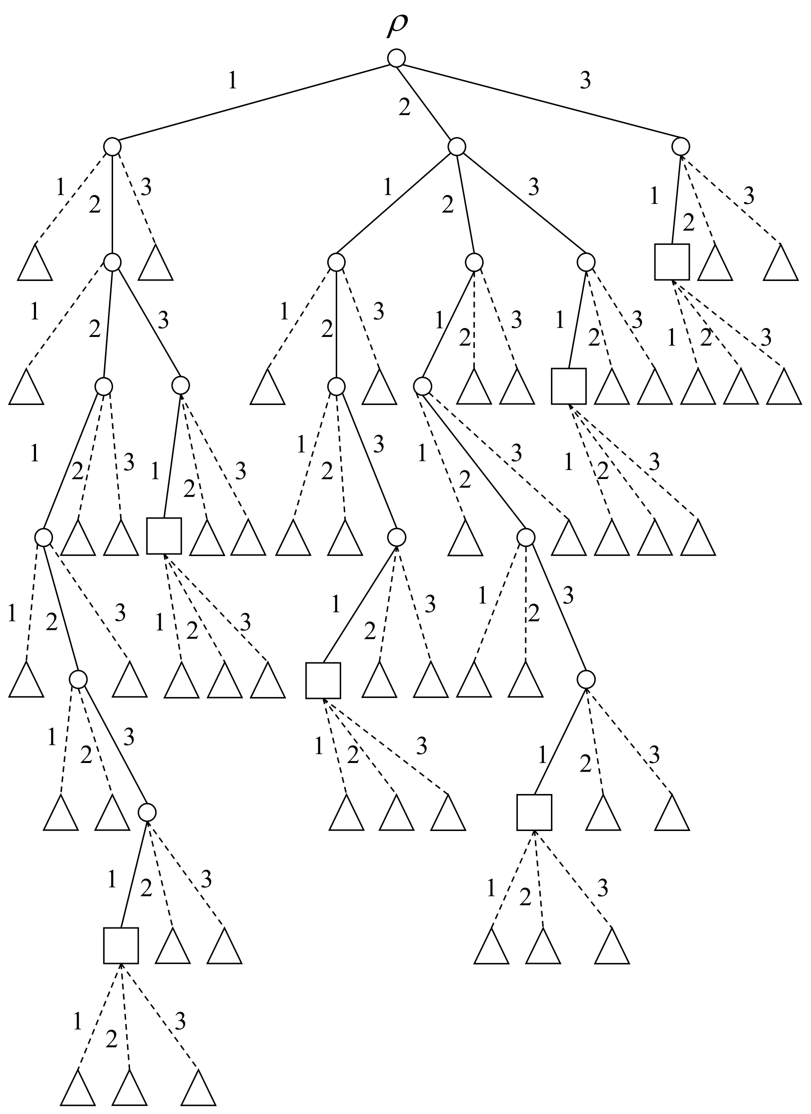


Figure 2.6: Extended trie $T_{ex}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

with a symbol in $\mathcal{X}$, and every path from ρ to a leaf corresponds to an MFW of $\mathcal{A}(\boldsymbol{x})$. The trie $T_{\mathcal{A}}(\boldsymbol{x})$ is a subtree of $T_{ex}(\boldsymbol{x})$. The leaf of $T_{\mathcal{A}}(\boldsymbol{x})$ is called *AD-node*. Figure 2.7 shows $T_{\mathcal{A}}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$, where $\mathcal{A}(\boldsymbol{x}) = \{11, 13, 32, 33, 121, 222, 223, 312, 2122\}$. In Figure 2.7, each triangle represents an AD-node.

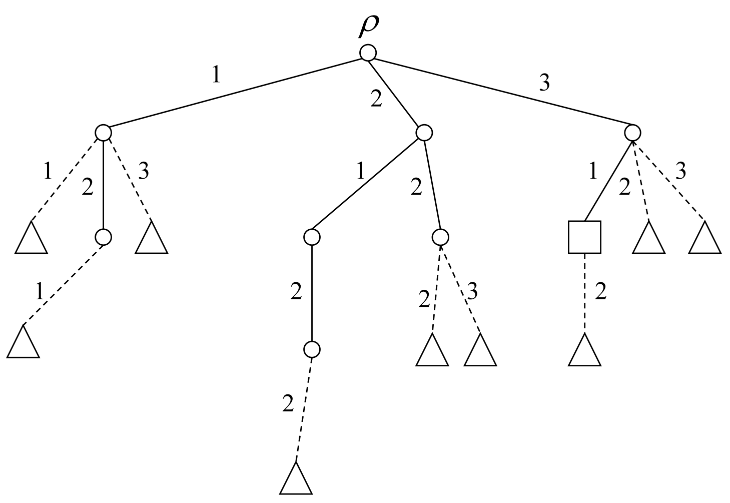


Figure 2.7: AD-trie $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

Morita *et al.* [MO05] proved the following Theorem 2 with respect to an AD-trie.

Theorem 2 (Morita and Ota, 2005). *Let q be a leaf in $\mathcal{T}(\boldsymbol{x})$, and let p be a child of q in $\mathcal{T}_{\mathsf{ex}}(\boldsymbol{x})$. Then, the path-string $\boldsymbol{w}(p)$ is an MFW if and only if the path from the root to q is the shortest one among all the leaves in $\mathcal{T}(\boldsymbol{x})$.*

Proof. This proof is omitted here. (see [MO05]) $\square$

From Theorem 1 and Theorem 2, we can classify $\mathcal{A}(\boldsymbol{x})$ into two classes. Elements of the first class are represented by $\boldsymbol{w}(q)a$, where $a \in \mathcal{X}$ and q is the leaf with the shortest path length among all the leaves of $\mathcal{T}(\boldsymbol{x})$. The set of the first class is denoted by $\mathcal{A}_L(\boldsymbol{x})$, and the AD-trie of $\mathcal{A}_L(\boldsymbol{x})$ is denoted by $\mathcal{T}_{\mathcal{A}_L}(\boldsymbol{x})$.

On the other hand, an element of the second class can be represented by $\boldsymbol{w}(p)b$, where p is an internal node of $\mathcal{T}(\boldsymbol{x})$ and $b \in \mathcal{X}$. The set of the second class is denoted by $\mathcal{A}_I(\boldsymbol{x})$, and the AD-trie of $\mathcal{A}_I(\boldsymbol{x})$ is denoted by $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$. For example, $\mathcal{A}_L(\boldsymbol{x})$ and $\mathcal{A}_I(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$ is given by

$$\{312\} \tag{2.11}$$

and

$$\{11, 13, 32, 33, 121, 222, 223, 2122\}, \tag{2.12}$$

respectively. Mignosi *et al.* [MRS02] and Morita *et al.* [MO05] has been investigated the upper bound on the size of $\mathcal{A}(\boldsymbol{x})$, independently. Table 2.1 shows the relationship between types of nodes in $\mathcal{T}(\boldsymbol{x})$ and classes of the antidictionary $\mathcal{A}(\boldsymbol{x})$.

Table 2.1: Classes of the antidictionary $\mathcal{A}(\boldsymbol{x})$.

node type of $\mathcal{T}(\boldsymbol{x})$	path-string	class	upper bound
internal	$\boldsymbol{w}(p)a$	$\mathcal{A}_I(\boldsymbol{x})$	$(L_n - 1)(n - 1)$
leaf	$\boldsymbol{w}(q)b$	$\mathcal{A}_L(\boldsymbol{x})$	L_n
all	$\boldsymbol{w}(p)a, \boldsymbol{w}(q)b$	$\mathcal{A}(\boldsymbol{x})$	$(L_n - 1)(n - 1) + L_n$

From Theorem 1 and Theorem 2 and computational complexity of the algorithm MF-Trie [CMR98] (the detail is described in Section 2.5.1 in this thesis.) to produce $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ from $\mathcal{G}(\boldsymbol{x})$, we obtain the next theorem.

Theorem 3. *An AD-trie $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ has at most $(L_n + 1)n$ nodes, where L_n is the number of all symbols occurring in the string $\boldsymbol{x}$ of length n.*

Proof. Let S_1 be the total number of states in $\mathcal{G}(\boldsymbol{x})$, and let S_2 be the total number of nodes that are created by the MF-Trie algorithm [CMR98]. The total number of nodes of $T_{\mathcal{A}}(\boldsymbol{x})$, denoted by $T(n)$, can be expressed by

$$T(n) = S_1 + S_2. \tag{2.13}$$

From Theorem 6.1 [CR02], S_1 is less than $2n$. Therefore, we obtain

$$S_1 \leq 2n - 1. \tag{2.14}$$

Since new nodes created by the MF-Trie algorithm are equal to the total number of elements of $\mathcal{A}(\boldsymbol{x})$, from Table 2.1, we obtain

$$S_2 \leq (L_n - 1)(n - 1) + L_n. \tag{2.15}$$

Therefore, from (2.13), (2.14) and (2.15), we obtain

$$\begin{aligned} T(n) &\leq S_1 + S_2 \qquad (2.16)\\ &\leq 2n - 1 + (L_n - 1)(n - 1) + L_n \\ &= (L_n + 1)n. \end{aligned}$$

□

2.5 Traditional Construction of an AD-Trie

There are two well-known construction algorithms for an AD-trie, and both of them were proposed by Crochemore *et al.* The first one is the algorithm MF-Trie using a suffix dawg, and it can construct the AD-trie of the antidictionary of a give string over any finite alphabet with $O(n)$ computational time with respect to the string length n [CMR98].

The second one is the algorithm S2ADB using a suffix trie. It can construct the AD-trie of the antidictionary of a given *binary* string, but requires $O(n^2)$ time and space [CMRS00]. The suffix trie makes the algorithm more easily understandable than the suffix dawg. Besides, it has the practical advantage in constructing a restricted antidictionary that is a subset of MFWs whose length are less than or equal to a given constant k. An implementation of this algorithm is given in [CMRS00] and it runs in $O(kn)$ time and space complexity while the algorithm needs an excessive amount of time and space as k grows. In Section 2.6, we generalize the S2ADB to any string over *finite* alphabet called the S2AD.

2.5.1 Suffix Dawgs

Crochemore *et al.* proposed the MF-Trie algorithm to convert $\mathsf{G}(\boldsymbol{x})$ into $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$ [CMR98]. The MF-Trie algorithm works in linear time and space with respect to the string length n. The outline of the MF-Trie is as follows.

```
Algorithm MF-Trie
    input  : a suffix dawg $\mathsf{G}(\boldsymbol{x})$
    output : the AD-trie $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$
begin                                                        1
  $\mathsf{T} \leftarrow \mathsf{G}(\boldsymbol{x})$;                          2
  for (each state $p$ of $\mathsf{T}$ in breadth-first order) do begin    3
    for (each symbol $a$ in $\mathcal{X}$) do begin                      4
```

```
      if ($(p, a)$ undefined in $\mathsf{G}(\boldsymbol{x})$ and $(\sigma(p), a)$ defined in $\mathsf{G}(\boldsymbol{x})$)
        $q \leftarrow$ new AD-node; $(p, a) \leftarrow q$ in $\mathsf{T}$;
      else if($(p, a) = r$ and $r$ already traversed in $\mathsf{G}(\boldsymbol{x})$) then
        remove an edge from $p$ to $r$ in $\mathsf{T}$;
    end for;
  end for;
  return $\mathsf{T}(= \mathsf{T}_{\mathcal{A}}(\boldsymbol{x}))$;
end.
```

2.5.2 Suffix Tries of Binary Strings

Crochemore *et al.* also proposed the construction algorithm of $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$ for a given binary string $\boldsymbol{x}$ using $\mathsf{T}(\boldsymbol{x})$. The algorithm called S2ADB produces $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$ from $\mathsf{T}(\boldsymbol{x})$ of a binary string $\boldsymbol{x}$ via the intermediate trie $\mathsf{T}_{\mathsf{I}}(x)$. The trie $\mathsf{T}_{\mathsf{I}}(x)$ stores all elements of both $\mathcal{D}(\boldsymbol{x})$ and $\mathcal{A}(\boldsymbol{x})$ of a binary string $\boldsymbol{x}$. In other words, both $\mathsf{T}(\boldsymbol{x})$ and $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$ are subtrees of $\mathsf{T}_{\mathsf{I}}(x)$. Moreover, $\mathsf{T}_{\mathsf{I}}(x)$ is a subtree of $\mathsf{T}_{\mathsf{ex}}(x)$. The trie $\mathsf{T}_{\mathsf{I}}(x)$ for $\boldsymbol{x} = 12122$ is shown in Figure 2.8, where $\mathcal{A}(\boldsymbol{x}) = \{11, 221, 222, 2121\}$.

The outline of the S2ADB is as follows.

```
Algorithm S2ADB
    input  : the suffix trie $\mathsf{T}(\boldsymbol{x})$ of a binary string $\boldsymbol{x}$
    output: the AD-trie $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$
begin
  $\mathsf{T} \leftarrow \mathsf{T}(\boldsymbol{x})$; $\mathcal{Q} \leftarrow \phi$;
```

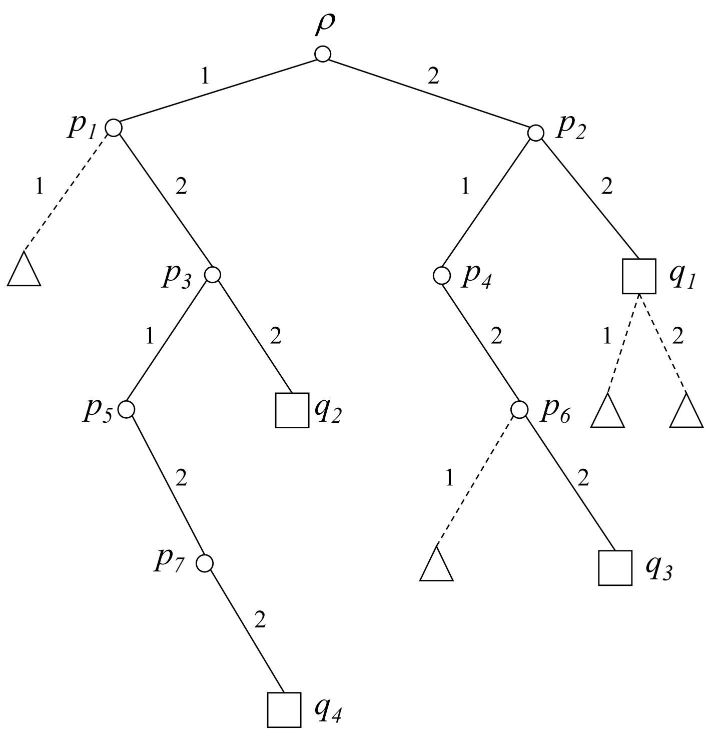


Figure 2.8: Intermediate trie $T_I(x)$ for $\boldsymbol{x} = 12122$.

```
/* Step 1: construct the intermediate trie $T_I(x)$ */
for (each node $p$ in $T$ in breadth-first order) do begin
  for (each symbol $a$ in $\{1,2\}$) do begin
    if ($a \notin \mathcal{L}(p)$ and $a \in \mathcal{L}(\sigma(p))$)
      $q \leftarrow$ new leaf; $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{q\}$;
      $(p, a) \leftarrow q$;
    end if;
  end for;
end for;
/* Step 2: remove edges of the trie $T_I(\boldsymbol{x})$ */
for (each node $p$ in $T$ in depth-first order) do
```

```
    remove node of $\mathsf{T}$ from which no path leads to $\mathcal{Q}$;
  **return** $\mathsf{T}(= \mathsf{T}_{\mathcal{A}}(\boldsymbol{x}))$;
**end**.
```

Since $\mathsf{T}(\boldsymbol{x})$ has $O(n^2)$ nodes with respect to a string length n, the S2ADB algorithm needs quadratic computational time and space.

To reduce computational time and space, Crochemore *et al.* also proposed the algorithm, called Build-Fact [CMRS00], to construct the restricted suffix trie that stores substrings of $\boldsymbol{x}$ whose length are less than or equal to a given constant k. By using the restricted suffix trie instead of $\mathsf{T}(\boldsymbol{x})$, the S2ADB works with $O(kn)$ time and space.

2.6 Construction of an AD-trie Using a Suffix Trie for Finite Alphabet

The S2ADB algorithm can construct only the AD-trie for only a binary string. In this section, we generalize the S2ADB to any string over finite alphabet. By applying Theorem 1, we can obtain a new algorithm S2AD to construct the AD-trie for a given string over finite alphabet based on the S2ADB. In other words, we can obtain the S2AD algorithm by exchanging binary alphabet $\{1, 2\}$ with finite alphabet $\mathcal{X}$ in the S2ADB.

The S2AD algorithm also produces $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$ from $\mathsf{T}(\boldsymbol{x})$ via $\mathsf{T}_I(x)$. The trie $\mathsf{T}_I(x)$ for $\boldsymbol{x} = 1221231$ is shown in Figure 2.9.

The outline of the S2AD is as follows.

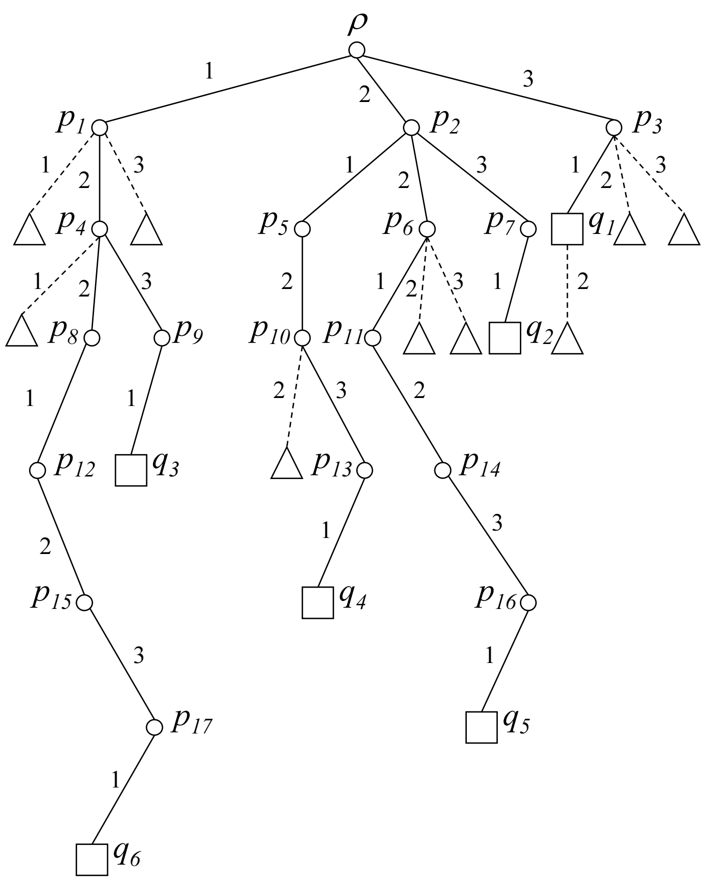


Figure 2.9: The intermediate trie $T_I(x)$ for $\boldsymbol{x} = 1221231$.

```
Algorithm S2AD
    input  : a suffix trie $T(\boldsymbol{x})$
    output : the AD-trie $T_{\mathcal{A}}(\boldsymbol{x})$
begin
  $T \leftarrow T(\boldsymbol{x})$; $\mathcal{Q} \leftarrow \phi$;
  /* construct the intermediate trie $T_I(x)$ */
  for (each node $p$ in $T$ in breadth-first order) do begin
    for (each symbol $a$ in $\mathcal{X}$) do begin
```

```
        if ($a \notin \mathcal{L}(p)$ and $a \in \mathcal{L}(\sigma(p))$)
          $q \leftarrow$ new leaf; $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{q\}$;
          $(p, a) \leftarrow q$;
        end if;
      end for;
    end for;
    /* remove edges of $\mathsf{T}_I(x)$ */
    for (each node $p$ in $\mathsf{T}$ in depth-first order) do
      remove node of $\mathsf{T}$ from which no path leads to $\mathcal{Q}$;
    return $\mathsf{T}(= \mathsf{T}_{\mathcal{A}}(\boldsymbol{x}))$;
end.
```

Chapter 3

Review of Data Compression Using Antidictionaries (DCA)

3.1 Introduction

The application of antidictionary of a binary string to lossless off-line data compression was first introduced by Crochemore *et al.* [CMRS00]. Their method, called Data Compression using Antidictionaries (DCA), was applied to the Calgary Corpus [Cal] that is commonly used for comparing data compression algorithms and it was shown that the DCA works as well as dictionary compression methods such as the Lempel-Ziv algorithm. They also proved that the DCA method achieves a compression rate that is equal to the entropy rate of a balanced binary source which is a *unifilar* Markov source with two output symbols of equal probabilities. A Markov source is called unifilar if for each state s, the labeled symbols of the transition from s to $s_1, \ldots, s_k$ are distinct, where $s_1, \ldots, s_k$ are the states that can be reached in one step from s [Ash90].

In 2004, we proposed an on-line source coding scheme using an antidictionary for lossless compression of electrocardiograms (ECGs) [OM04]. Experimental results showed that the proposed algorithm achieved a compression ratio that was about 10% better than the compression ratio achieved by the LZ algorithms. The details of the proposed algorithm for ECG will be described in Chapter 6. In the last two or three years, extensions of the DCA algorithm to any string over finite alphabet were proposed [CEGM04, OM06c]. The DCA algorithm and their extensions need an antidictionary of a given string and those encoders to compress a given string. The construction algorithms of an antidictionary in linear time and space were proposed [CMR98, OM06b, OM06c, OM07b]. The details are described in Chapter 4.

In this Chapter 3, we introduce the basic and practical schemes of the DCA. We present an arithmetic coding based on the DCA for binary strings proposed by Ohkawa *et al.* [OHY05], and we generalize the OHY to any string over finite alphabet.

3.2 Encoding

The set $\mathcal{V}_i(\boldsymbol{x})$ is defined as the subset of $\mathcal{A}_I(\boldsymbol{x})$, that is,

$$\mathcal{V}_i(\boldsymbol{x}) = \{\boldsymbol{v} | \boldsymbol{v} = \boldsymbol{u}a \in \mathcal{A}_I(\boldsymbol{x}), \boldsymbol{u} \in \mathcal{S}(\boldsymbol{x}^i), a \in \mathcal{X}\}. \tag{3.1}$$

For an arbitrary i, if $|\mathcal{V}_i(\boldsymbol{x})| = L_n - 1$, then a symbol x_{i+1} is eliminated in the DCA algorithm. In Eq. (3.1), we use $\mathcal{A}_I(\boldsymbol{x})$ instead of $\mathcal{A}(\boldsymbol{x})$ because elements of $\mathcal{A}_L(\boldsymbol{x})$ are useless for eliminating symbols. No symbol follows the string $\boldsymbol{u}$ such that $\boldsymbol{u}a \in \mathcal{A}_L(\boldsymbol{x})$ since $\boldsymbol{u}$ is a suffix of $\boldsymbol{x}$.

In other words, suppose to have just read proper prefix $\boldsymbol{x}^i$ of $\boldsymbol{x}$. If the string $\boldsymbol{u}a \in \mathcal{A}_I(\boldsymbol{x})$, such that $\boldsymbol{u}$ is a suffix of $\boldsymbol{x}^i$ and $a \in \mathcal{L}(\rho)$, then symbol x_{i+1} is not the symbol a. Hence, if there exists a string $\boldsymbol{u}a \in \mathcal{A}_I(\boldsymbol{x})$ such that $\boldsymbol{u} \in \mathcal{S}(\boldsymbol{x}^i)$ for every symbol $a \in \mathcal{L}(\rho)\backslash\{b\}$, then the symbol x_{i+1} is surely the symbol b because x_{i+1} is not the symbol $a \in \mathcal{L}(\rho)\backslash\{b\}$. We know in advance the next symbol x_{i+1} that turns out to be redundant or predictable.

Next, we present the encoding algorithm DCA Encoder of the DCA algorithm. For a given string $\boldsymbol{x}$ of length n, let $\boldsymbol{\gamma}$ be the encoded string of $\boldsymbol{x}$. The encoding algorithm DCA Encoder is as follows. Note that we use $\mathcal{A}_I(\boldsymbol{x})$ instead of $\mathcal{A}(\boldsymbol{x})$ in the following algorithms since $\mathcal{A}_L(\boldsymbol{x})$ is useless for eliminating symbols. In other words, the DCA algorithm using $\mathcal{A}_I(\boldsymbol{x})$ outputs the same γ as that of the DCA algorithm using $\mathcal{A}(\boldsymbol{x})$.

Algorithm DCA Encoder

```
    input  : a 2-tuples (an input string $\boldsymbol{x}$, $\mathcal{A}_I(\boldsymbol{x})$)
    output : the 3-tuples (the encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
begin                                                        1
  $\boldsymbol{\gamma} \leftarrow \lambda$;                  2
  for $i := 1$ to $n$ do begin                               3
    if $(|\mathcal{V}_i(\boldsymbol{x})| < L_n - 1)$ then    4
      $\boldsymbol{\gamma} \leftarrow \gamma.x_i$;           5
    end for;                                   6
    return ($\gamma$, $|\boldsymbol{x}|$, $\mathcal{A}_I(\boldsymbol{x})$);              7
end.                                           8
```

For example, Table 3.1 shows the relationship between output symbols of the DCA Encoder and $|\mathcal{V}_i(\boldsymbol{x})|$, where for $\boldsymbol{x} = 1221231$, $\mathcal{A}_I(\boldsymbol{x}) = \{11, 13, 32, 33, 121, 222, 223, 2122\}$, and $\mathcal{L}(\rho) = \{1, 2, 3\}$. As shown in Table 3.1, a symbol can be eliminated if $|\mathcal{V}_i(\boldsymbol{x})| = 2$ since $L_6 = 3$. As a result, the algorithm DCA Encoder obtains the 3-tuples (12, 7, $\mathcal{A}_I(\boldsymbol{x})$) as its codeword.

Table 3.1: An example of encoding of the DCA algorithm, where $\boldsymbol{x} = 122132$, $\mathcal{A}_I(\boldsymbol{x}) = \{11, 23, 31, 33, 121, 212, 222, 2122\}$, and $L_6 = 3$.

input string	output string	$\lvert\mathcal{V}_i(\boldsymbol{x})\rvert$	MFWs
$\boldsymbol{x}^0 = \lambda$	λ		
$\boldsymbol{x}^1 = 1$	1	0	
$\boldsymbol{x}^2 = 12$	1	2	$11, 13$
$\boldsymbol{x}^3 = 122$	12	1	121
$\boldsymbol{x}^4 = 1221$	12	2	$222, 223$
$\boldsymbol{x}^5 = 12212$	12	2	$11, 13$
$\boldsymbol{x}^6 = 122123$	12	2	$121, 2122$
$\boldsymbol{x}^7 = 1221231 (= \boldsymbol{x})$	$12 (= \gamma)$	2	$32, 33$

3.3 Decoding

Suppose that the string $\boldsymbol{v} (= \boldsymbol{x}^i)$ is already decoded string in the decoding algorithm. Let $\boldsymbol{\gamma}$ be the encoded string of $\boldsymbol{x}$ by the DCA Encoder. The decoding process is as follows. Let b be the symbol following the string $\boldsymbol{v}$. If $|\mathcal{V}_i(\boldsymbol{v})| = L_n - 1$, then the symbol b is surely the uniquely symbol a such that

$a \in \mathcal{L}(\rho)$ and $\boldsymbol{u}a \notin \mathcal{A}_I(\boldsymbol{x})$ for every $\boldsymbol{u} \in \mathcal{S}(\boldsymbol{v})$. Otherwise, $|\mathcal{V}_i(\boldsymbol{v})| < L_n - 1$, then the symbol b is read from the encoded string $\boldsymbol{\gamma}$.

The decoding algorithm DCA Decoder is as follows.

```
Algorithm DCA Decoder
    input  : a 3-tuples (an encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
    output: the string $\boldsymbol{y}(=\boldsymbol{x})$ of length $n$
begin
  $y_1 \leftarrow \gamma_1$; $j \leftarrow 2$;
  for $i := 2$ to $n$ do begin
    if $(|\mathcal{V}_{i-1}(\boldsymbol{y})| = L_n - 1))$
      $a \leftarrow$ the symbol $b$ such that $b \in \mathcal{L}(\rho)$ and $\boldsymbol{u}b \notin \mathcal{A}_I(\boldsymbol{x})$
                for every $\boldsymbol{u} \in \mathcal{S}(\boldsymbol{y}^{i-1}))$
      $y_i \leftarrow a$;
    else /* $|\mathcal{V}_{i-1}(\boldsymbol{y})| < L_n - 1$ */
      $y_i \leftarrow \gamma_j$; $j \leftarrow j + 1$;
  end for;
  return $\boldsymbol{y}$;
end.
```

To examine effectively whether or not the string $\boldsymbol{u}a$ is a element of $\mathcal{A}_I(\boldsymbol{x})$, a finite deterministic automaton, called *AD-automaton*, is used. By using the AD-automaton of the string $\boldsymbol{x}$, the DCA Encoder and DCA Decoder can be implemented to run in $O(n)$ time with respect to the string length n. Next Section details an AD-automaton.

3.4 AD-automatons

The DCA algorithm requires $|\mathcal{V}_i(\boldsymbol{x})|$ for every $\boldsymbol{x}^i$ in its encoding and decoding process. To obtain $|\mathcal{V}_i(\boldsymbol{x})|$ efficiently, an *AD-automaton* is used in the DCA algorithm [CMRS00]. The AD-automaton is produced from an AD-trie. Let $G_{\mathcal{A}}(\boldsymbol{x})$ be the AD-automaton of $\mathcal{A}(\boldsymbol{x})$ and $G_{\mathcal{A}_I}(\boldsymbol{x})$ be the AD-automaton of $\mathcal{A}_I(\boldsymbol{x})$. For example, Figure 3.1 and Figure 3.2 shows the AD-trie $T_{\mathcal{A}_I}(\boldsymbol{x})$ and the AD-automaton $G_{\mathcal{A}_I}(\boldsymbol{x})$ of $\mathcal{A}_I(\boldsymbol{x}) = \{11, 13, 32, 33, 121, 222, 223, 2122\}$ for $\boldsymbol{x} = 1221231$, respectively.

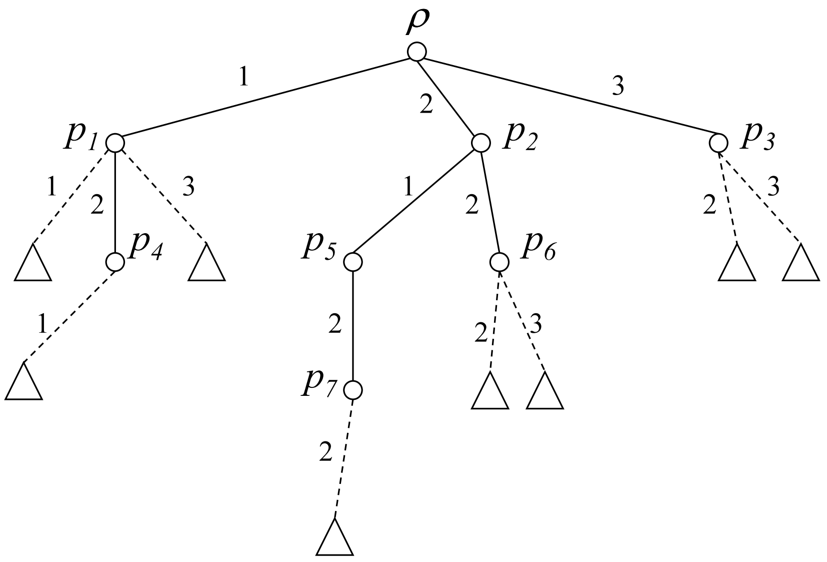


Figure 3.1: AD-trie $T_{\mathcal{A}_I}(\boldsymbol{x})$ for $\mathcal{A}_I(\boldsymbol{x}) = \{11, 13, 32, 33, 121, 222, 223, 2122\}$ for $\boldsymbol{x} = 1221231$.

In Figure 3.2, a solid line and a dashed line represents *possible-transition* and *impossible-transition*, respectively. A circle and a triangle represents *terminal* state and *non-terminal* state called *AD-state*, respectively. A terminal state corresponds to an internal node in $T_{\mathcal{A}_I}(\boldsymbol{x})$, and an AD-state corre-

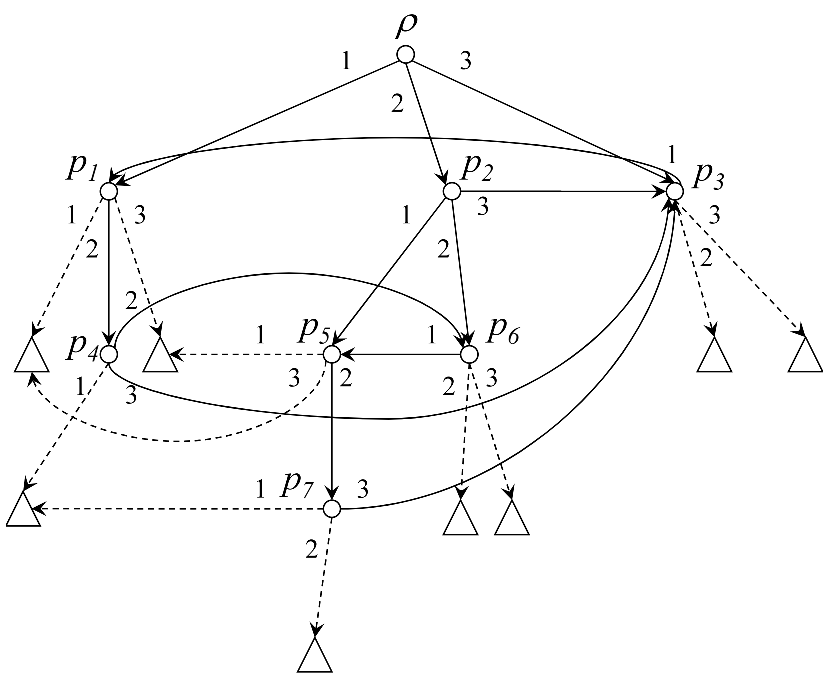


Figure 3.2: AD-automaton $G_{\mathcal{A}_I}(\boldsymbol{x})$ for $\mathcal{A}_I(\boldsymbol{x}) = \{11, 13, 32, 33, 121, 222, 223, 2122\}$ for $\boldsymbol{x} = 1221231$.

sponds to a leaf, that is an AD-node, in $T_{\mathcal{A}_I}(\boldsymbol{x})$. In Figure 3.2, $G_{\mathcal{A}_I}(\boldsymbol{x})$ has eight terminal states $(\rho, p_1, ..., p_7)$ and eight AD-states. The initial state of $G_{\mathcal{A}_I}(\boldsymbol{x})$ that corresponds to ρ of $T_{\mathcal{A}_I}(\boldsymbol{x})$ is also denoted by ρ. An impossible-transition is a transition toward an AD-state. Any impossible-transition corresponds to an MFW occurring in $\boldsymbol{x}$. Hence, the DCA algorithm allows possible-transitions.

Let $\mathcal{W}_i(\boldsymbol{x})$ be the set of common substrings between proper prefixes of MFWs in $\mathcal{A}_I(\boldsymbol{x})$ and suffixes of $\boldsymbol{x}^i$, that is,

$$\mathcal{W}_i(\boldsymbol{x}) = \{\boldsymbol{w} | \boldsymbol{w}\boldsymbol{v} \in \mathcal{A}_I(\boldsymbol{x}), \boldsymbol{w} \in \mathcal{S}(\boldsymbol{x}^i), \boldsymbol{v} \in \mathcal{X}^+\} \cup \{\lambda\}. \tag{3.2}$$

Let ξ be the function to obtain the longest string in a given set of strings. The string $\boldsymbol{w}_i$ is written as a function ξ such that

$$\boldsymbol{w}_i = \xi(\mathcal{W}_i(\boldsymbol{x})). \tag{3.3}$$

By using $G_{\mathcal{A}_I}(\boldsymbol{x})$, we can obtain $|\mathcal{V}_i(\boldsymbol{x})|$ in linear time using locus π_i of $\boldsymbol{w}_i$ in $T_{\mathcal{A}_I}(\boldsymbol{x})$ such that

$$\pi_i = l(\boldsymbol{w}_i). \tag{3.4}$$

If π_i has $L_n - 1$ impossible-transitions, then $|\mathcal{V}_i(\boldsymbol{x})| = L_n - 1$. As an example, for $\boldsymbol{x} = 1221231$, the longest string $\boldsymbol{s}_4$ in $\mathcal{W}_4(\boldsymbol{x})$ is 21, and the locus π_i is q_5 in Figure 2.7, where $\boldsymbol{x}^4 = 1221$.

For example, we show encoding process for $\boldsymbol{x} = 1221231$ using $G_{\mathcal{A}_I}(\boldsymbol{x})$ in Figure 3.2. Starting from ρ of $G_{\mathcal{A}_I}(\boldsymbol{x})$, transitions of $G_{\mathcal{A}_I}(\boldsymbol{x})$ with $\boldsymbol{x} = 1221231$ will be given as follows;

$$\rho \xrightarrow{1} p_1 \xrightarrow{2} p_4 \xrightarrow{2} p_6 \xrightarrow{1} p_5 \xrightarrow{2} p_7 \xrightarrow{3} p_3 \xrightarrow{1} p_1. \tag{3.5}$$

As shown in Figure 3.2, in (3.5), outputs occur at state ρ and p_4 since these states have less than $L_n - 1$ impossible-transition. Outputs never occur at any node except ρ and p_4 since each state has $L_n - 1$ impossible-transitions. Table 3.2 shows the relationship between the number of impossible-transitions at each state and $|\mathcal{V}_i(\boldsymbol{x})|$ in encoding process.

Crochemore *et al.* proposed the algorithm L-Automaton to convert $T_{\mathcal{A}}(\boldsymbol{x})$ into $G_{\mathcal{A}}(\boldsymbol{x})$ [CMRS00]. This algorithm works in linear time from Theorem 3 with respect to the string length.

Table 3.2: The relationship between the number of impossible-transitions and $|\mathcal{V}_i(\boldsymbol{x})|$, where $\boldsymbol{x} = 122132$, $\mathcal{A}_I(\boldsymbol{x}) = \{11, 23, 31, 33, 121, 212, 222\}$, and $L_6 = 3$.

input string	output string	states	impossible-transitions	$\lvert\mathcal{V}_i(\boldsymbol{x})\rvert$	MFWs
$\boldsymbol{x}^0 = \lambda$	λ		0		
$\boldsymbol{x}^1 = 1$	1	ρ	0	0	
$\boldsymbol{x}^2 = 12$	1	p_1	2	2	$11, 13$
$\boldsymbol{x}^3 = 122$	12	p_4	1	1	121
$\boldsymbol{x}^4 = 1221$	12	p_6	2	2	$222, 223$
$\boldsymbol{x}^5 = 12212$	12	p_5	2	2	$11, 13$
$\boldsymbol{x}^6 = 122123$	12	p_7	2	2	$121, 2122$
$\boldsymbol{x}^7 = 1221231(= \boldsymbol{x})$	$12(= \gamma)$	p_3	2	2	$32, 33$

By using an AD-automaton, the DCA Encoder and DCA Decoder can be implemented to run in $O(n)$ time with respect to the string length n. If the state p for $\boldsymbol{x}^i$ has just $L_n - 1$ impossible-transitions, then $|\mathcal{V}_i(\boldsymbol{x}^i)| = L_n - 1$. We show the encoding and decoding algorithm using $G_{\mathcal{A}_I}(\boldsymbol{x})$, respectively. The notation (p, a) denotes the transition from a state p with a symbol a. The outline of the DCA Encoder-AU and DCA Decoder-AU are as follows, respectively.

Algorithm DCA Encoder-AU

```
    input  : a 2-tuples (a string $\boldsymbol{x}$ of length $n$, $\mathcal{A}_I(\boldsymbol{x})$)
    output : the 3-tuples (the encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
begin
  /* Step 1:Construct $G_{\mathcal{A}_I}(\boldsymbol{x})$ from $T_{\mathcal{A}_I}(\boldsymbol{x})$ */
  $G_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{x})$);
  /* Step 2:Encode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{x})$;
  **for** $i := 1$ **to** $n$ **do begin**
    **if** ($p$ has less than $L_n - 1$ impossible-transitions) **then**
      $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma}.x_i$;
    $p \leftarrow (p, x_i)$;
  **end for**;
  **return** ($\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$);
**end**.
```

Algorithm DCA Decoder-AU

```
    input  : a 3-tuples (an encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
    output : the string $\boldsymbol{y}(= \boldsymbol{x})$ of length $n$
**begin**
  /* *Step 1:Reconstruct* $G_{\mathcal{A}_I}(\boldsymbol{x})$ *from* $T_{\mathcal{A}_I}(\boldsymbol{x})$ */
  $G_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{x})$);
  /* *Step 2:Decode* */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{x})$; $\boldsymbol{y} \leftarrow \lambda$;
  **for** $i := 1$ **to** $n$ **do begin**
    **if** ($p$ has less than $L_n - 1$ impossible-transitions) **then begin**
      $y_i \leftarrow \gamma_j$; $j \leftarrow j + 1$;
    **else**
      $y_i \leftarrow a \in \mathcal{L}(p)$;
    **end if**;
    $p \leftarrow (p, y_i)$;
  end for;
  return $\boldsymbol{y}(=\boldsymbol{x})$;
end.
```

3.5 Practical Techniques

In this section, we introduce some techniques to improve compression ratio of the DCA algorithm [CMRS00].

3.5.1 Recursive Tree Representation

An encoder of the DCA algorithm needs to send an antidictionary to its decoder. We need a compact representation of the antidictionary.

Figure 3.3 shows $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ of $\mathcal{A}(\boldsymbol{x}) = \{11, 221, 222, 2121\}$. As shown in Figure 3.3, the MFWs 222 and 221 share the common prefix 22. To use the common prefix among MFWs efficiently, we apply a recursive tree representation to $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$. Assuming that $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ is a binary tree. The node in a binary tree has two subtrees, only the right subtree, only the left subtree or no subtree. The node can be represented by the string 11, 10, 01, 00, respectively. The whole tree can be encoded by traversing the depth-first order. A given binary tree that has n nodes can be represented by $2n$ bits since the cost of one node is just 2 bits. For example, $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ in Figure 3.3 is represented by the string 11100011011000110000, that is 20 bits, since $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ has 10 nodes. Usually, we use recursive tree representation of the AD-trie to store an antidictionary as a codeword of the DCA algorithm.

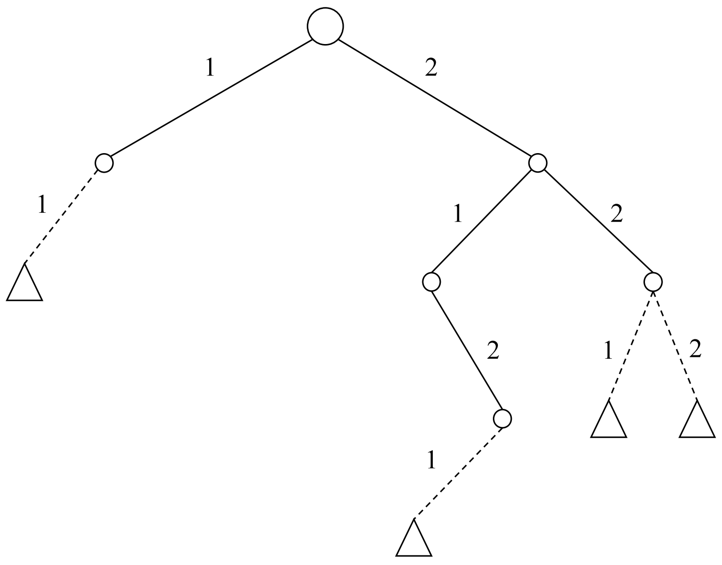


Figure 3.3: AD-trie $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ of $\mathcal{A}(\boldsymbol{x}) = \{11, 221, 222, 2121\}$.

3.5.2 Pruning an Antidictionary

From Theorem 3, since $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ for a binary string $\boldsymbol{x}$ of length n has at most $3n$ nodes, the cost of $\mathcal{A}(\boldsymbol{x})$ requires $6n$ bits in worst case. The cost of $\mathcal{A}(\boldsymbol{x})$ is larger than that of $\boldsymbol{x}$. To improve the size of the codeword, it needs to reduce the cost of $\mathcal{A}(\boldsymbol{x})$.

Let p, q be a node of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ such that $\boldsymbol{w}(p) \in \mathcal{A}(\boldsymbol{x})$ and the parent node p, respectively. Let $\boldsymbol{w}(q, p)$ be a symbol a. If $\boldsymbol{w}(q)$ appears k times in $\boldsymbol{x}$, then the MFW $\boldsymbol{w}(p)$ contributes elimination of k symbols in $\boldsymbol{x}$. On the other hand, the cost of $\boldsymbol{w}(p)$ in the codeword is $2|\boldsymbol{w}(p)|$ bits by using recursive tree representation. If $k < 2|\boldsymbol{w}(p)|$, then the cost is larger than the gain in compression. Otherwise, the gain is larger than the cost.

Let $c(p)$ be the number of occurrence of $\boldsymbol{w}(q)$ in $\boldsymbol{x}$. The function $c(\cdot)$ is called *cost function*. The MFW $\boldsymbol{w}(p)$ contributes elimination of $c(p)$ symbols in encoding process. Crochemore *et al.* proposed the algorithm to obtain $c(p)$ for each leaf p in $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ using $\mathcal{G}_{\mathcal{A}}(\boldsymbol{x})$ [CMRS00]. The size of $c(p)$ corresponds to the number of times that p is traversed in $\mathcal{G}_{\mathcal{A}}(\boldsymbol{x})$ while reading the string $\boldsymbol{x}$.

Crochemore *et al.* proposed the criteria function called *gain function* to select MFWs from $\mathcal{A}(\boldsymbol{x})$ [CMRS00]. The gain function $g(S)$ to obtain the gain of a subtree S of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ is defined by

$$g(S) = \begin{cases} 0 & \text{if } S \text{ is empty} \\ c(p) - 2 & \text{if } S \text{ is a leaf } p \\ g(S_1) - 2 & \text{if } S \text{ has one subtree } S_1 \\ M - 2 & \text{if } S \text{ has two subtrees } S_1 \text{ and } S_2, \end{cases} \tag{3.6}$$

where M is the maximum integer of three values $g(S_1), g(S_2)$ and $(g(S_1) + g(S_2))$.

Crochemore *et al.* proposed the algorithm called Simple Pruning to prune $\mathcal{A}(\boldsymbol{x})$. We will use the cost function $c(p)$ and the gain function $g(S)$ in the Simple Pruning. Let initial value of $c(p)$ be zero for all nodes of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$. The outline of the Simple Pruning is as follows.

```
Algorithm Simple Pruning
    input  : a 2-tuples (an AD-trie $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$, the function cost $c$)
    output : the pruned AD-trie $\mathcal{T}_{\mathcal{A}_p(\boldsymbol{x})}$
begin                                                                  1
  /* Step 1: compute $g(S)$ for each subtree $S$ of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ */          2
  for (each node $p$ in $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ in depth-first order) do         3
```

```
    compute $g(S)$; /* $p$ is the root of subtree $S$ */
  /* Step 2: eliminate subtree $S$ of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ for $g(S) \leq 0$ */
  for (each node $p$ in $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ in depth-first order) do
    if ($g(S) \leq 0$) then /* $p$ is the root of subtree $S$ */
      eliminate subtree $S$;
  return $\mathcal{T}_{\mathcal{A}_p}$
end.
```

3.5.3 Self-Compression

Crochemore *et al.* proposed the lossless compression algorithm for an antidictionary of a binary string [CMRS00]. This algorithm is called Self-Compress. The Self-Compress produces the compressed AD-trie from $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$. They also proposed the algorithm to construct $\mathcal{G}_{\mathcal{A}}(\boldsymbol{x})$ from the compressed AD-trie.

In 2004, the extension of Self-Compress to any string over finite alphabet was proposed [CEGM04].

3.6 Elementary of an Arithmetic Coding Based on the DCA

In 2005, Ohkawa *et al.* [OHY05] proposed the off-line compression algorithm for binary strings that uses an AD-automaton as a statistical model for an arithmetic coding. Simulation results showed that the AD-automaton for a binary string provides an efficient statistical model. Their method, called

Ohkawa-Harada-Yamamoto (OHY) method, constructs a statistical model such that a conditional probability for each state in the AD-automaton by means of the transition times of each state with a give binary string. The OHY algorithm combines eliminating symbols of a given string using an AD-automaton and encoding the remained symbols by means of an arithmetic coding for a binary string.

We now show the encoding algorithm OHY Encoder of the OHY method. Let $N(c,p)$ be the number of times that transitions from a state p with symbol $c \in \{1,2\}$ occurring in the encoding process.

The initial value of $N(c,p)$ is one for $c \in \{1,2\}$. Let $N(p)$ be the number of the total transition times from p, that is $N(p)$ is equal to $\sum_{c\in\{1,2\}} N(c,p)$. The algorithm uses the procedure AC-E $(c,p,\boldsymbol{s})$ to encode the symbol c by means of cumulative probabilities for an element of $\mathcal{L}(p)$ by an adaptive arithmetic coding, where the string $\boldsymbol{s}$ denotes a codeword. The outline of the OHY Encoder is as follows.

```
Algorithm OHY Encoder
    input  : a 2-tuples (a binary string $\boldsymbol{x}$ of length $n$, $\mathcal{A}_I(\boldsymbol{x})$)
    output : the 3-tuples (an encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
begin
  /* Step 1:Construct $G_{\mathcal{A}_I}(\boldsymbol{x})$ from $T_{\mathcal{A}_I}(\boldsymbol{x})$ */
  $G_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ L-Automaton($T_{\mathcal{A}}(\boldsymbol{x})$);
  /* Step 2:Encode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{x})$; $\boldsymbol{\gamma} \leftarrow \lambda$;
  for $i := 1$ to $n$ do begin
```

```
    if ($p$ has no impossible-transition) then begin
      $\boldsymbol{\gamma} \leftarrow$ AC-E $(x_i, p, \boldsymbol{\gamma})$; /* an adaptive arithmetic coding */
      $N(c,p) \leftarrow N(c,p) + 1$;
    end if;
    $p \leftarrow (p, x_i)$;
  end for;
  return ($\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$);
end.
```

Next, we show the OHY Decoder algorithm. The algorithm uses the procedure AC-D $(\boldsymbol{s}, p)$ to decode the symbol c by means of cumulative probabilities of $\{1, 2\}$ and the codeword $\boldsymbol{s}$ by an adaptive arithmetic coding. The outline of the OHY Decoder is as follows.

```
Algorithm OHY Decoder
    input  : a 3-tuples (an encoded string $\boldsymbol{\gamma}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{x})$)
    output : the string $\boldsymbol{y} (= \boldsymbol{x})$ of length $n$
begin
  /* Step 1:Reconstruct $G_{\mathcal{A}_I}(\boldsymbol{x})$ */
  $G_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{x})$);
  /* Step 2:Decode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{x})$; $\boldsymbol{y} \leftarrow \lambda$;
  for $i := 1$ to $n$ do begin
    if ($p$ has no impossible-transition) then begin
      $c \leftarrow$ AC-D $(\boldsymbol{\gamma}, p)$; /* an adaptive arithmetic coding */
      $\boldsymbol{y} \leftarrow \boldsymbol{y}.c$; $N(c,p) \leftarrow N(c,p)+1$;
    **end if**;
    **else**
      $c \leftarrow b \in \mathcal{L}(p)$; $\boldsymbol{y} \leftarrow \boldsymbol{y}.c$;
    $p \leftarrow (p,c)$;
  **end for**;
  **return** $\boldsymbol{y}(=\boldsymbol{x})$;
**end**.
```

3.7 An Arithmetic Coding Using an AD-automaton for Finite Alphabet

The OHY algorithm can compress only a given binary string. In this section, we generalize the OHY algorithm to any string over finite alphabet. The proposed algorithm ACDCA is obtained by combining the DCA algorithm for finite alphabet with an adaptive arithmetic coding.

First, we show the encoding algorithm of the ACDCA. The initial value of $N(c,p)$ is one for $c \in \mathcal{L}(p)$. Let $N(p)$ be the number of the total transition times from p. The size $N(p)$ is equivalent to $\sum_{c\in\mathcal{L}(p)} N(c,p)$. The algorithm uses the procedure AC-E $(c,p,\boldsymbol{s})$ to encode the symbol c by means of cumulative probabilities for an element of $\mathcal{L}(p)$ by an adaptive arithmetic coding, where the string $\boldsymbol{s}$ denotes a codeword.

The outline of the ACDCA-E is as follows.

Algorithm ACDCA-E

```
    input  : a 2-tuples (a string $\boldsymbol{x}$ of length $n$, $\mathcal{A}_I(\boldsymbol{x})$)
    output : the 3-tuples (the encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
begin
  /* Step 1:Construct $G_{\mathcal{A}_I}(\boldsymbol{x})$ from $T_{\mathcal{A}_I}(\boldsymbol{x})$ */
  $G_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{x})$);
  /* Step 2:Encode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{x})$; $\boldsymbol{\gamma} \leftarrow \lambda$;
  for $i := 1$ to $n$ do begin
    if ($p$ has less than $L_n - 1$ impossible-transitions) then begin
      $\boldsymbol{\gamma} \leftarrow$ AC-E $(x_i, p, \boldsymbol{\gamma})$; /* an adaptive arithmetic coding */
      $N(c,p) \leftarrow N(c,p) + 1$;
    end if;
    $p \leftarrow (p, x_i)$;
  end for;
  return ($\boldsymbol{\gamma}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{x})$);
end.
```

Next, we show the decoder of the ACDCA algorithm. The algorithm uses the procedure AC-D $(\boldsymbol{s}, p)$ to decode the symbol c by means of cumulative probabilities of $\mathcal{L}(p)$ and the codeword $\boldsymbol{s}$ by an arithmetic coding. The outline of the ACDCA-D is as follows.

Algorithm ACDCA-D

```
  input  : a 3-tuples (an encoded string $\boldsymbol{\gamma}$, $n$, $\mathcal{A}_I(\boldsymbol{x})$)
  output : the string $\boldsymbol{y}(=\boldsymbol{x})$ of length $n$
begin
  /* Step 1:Reconstruct $G_{\mathcal{A}_I}(\boldsymbol{x})$ */
  $G_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{x})$);
  /* Step 2:Decode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{x})$; $\boldsymbol{y} \leftarrow \lambda$;
  for $i := 1$ to $n$ do begin
    if ($p$ has less than $L_n - 1$ impossible-transitions) then begin
      $c \leftarrow$ AC-D $(\boldsymbol{\gamma}, p)$; /* an adaptive arithmetic coding */
      $\boldsymbol{y} \leftarrow \boldsymbol{y}.c$;
      $N(c,p) \leftarrow N(c,p) + 1$;
    end if;
    else
      $c \leftarrow b \in \mathcal{L}(p)$; $\boldsymbol{y} \leftarrow \boldsymbol{y}.c$;
    $p \leftarrow (p, c)$;
  end for;
  return $\boldsymbol{y}(=\boldsymbol{x})$;
end.
```

Chapter 4

Construction of an Antidictionary with Linear Computational Complexity

4.1 Introduction

The antidictionary for a given string is the set of all minimal strings that never appear in the string. The antidictionary is useful for data compression and fragment assembly problem [CMRS00, OM04, FMRS06]. Morita *et al.* proposed an algorithm to reconstruct the suffix trie from an AD-trie [MO05]. In 2006, Sun *et al.* proposed an application of the antidictionary of a binary string to synchronization markers in video stream [SMN06].

There are two well-known construction algorithms for an antidictionary [CMR98, CMRS00]. The first one is the algorithm using a suffix dawg, and it can construct the antidictionary of a given string over any finite alphabet with $O(n)$

computational time with respect to the string length n [CMR98].

The second one is the algorithm using a suffix trie. It can construct the antidictionary of a given binary string and requires $O(n^2)$ time and space with respect to the string length n [CMRS00]. The suffix trie makes the algorithm more easily understandable than the suffix dawg. Besides, it has a practical advantage in constructing a restricted antidictionary that is a subset of MFWs whose length is less than or equal to a given constant k. An implementation of this algorithm is given in [CMRS00] and it runs in $O(kn)$ time and space complexity, while the algorithm needs an excessive amount of time and space as k grows.

To reduce the computational complexity for construction of the antidictionary using a suffix trie, we can use a suffix tree. However, even if we apply a suffix tree, the time complexity of direct construction algorithms for the antidictionary remains $O(n^2)$, because there are $O(n^2)$ path-strings for all leaves in the extended trie for which one needs to examine whether or not they are MFWs.

The main result of this chapter is the linear algorithm to construct an antidictionary of a string over finite alphabet using a suffix tree [OM05a, OM05b, OM06b, OM06c, OM07b]. We introduce a new concept that uses a new kind of pointers, called *MF-links*, to efficiently traverse through the suffix tree. The MF-links are as essential as the suffix links for construction of an antidictionary in linear computational time. We prove that the total construction computational complexity can be reduced to $O(n)$ with respect to the string length n.

Moreover, we propose an algorithm to construct an AD-trie for a given string using a suffix tree, since the antidictionary is represented by recursive tree representation of an AD-trie in the DCA algorithm. It is proved that the construction algorithm of an AD-trie also works in linear time.

This chapter is organized as follows. In Section 4.2, we introduce the new concept of MF-links and derive a proposition that we will use to construct an antidictionary efficiently. Then, this section details the proposed algorithm for construction of the antidictionary using a suffix tree, and it is proved that the proposed algorithm has a linear time complexity. Finally, its effectiveness is demonstrated by simulation results. In Section 4.3, we propose an algorithm to construct the AD-trie for a given string using a suffix tree in linear time. Section 4.4 summarizes our results.

4.2 Construction of an Antidictionary Using a Suffix Tree in linear time

4.2.1 MF-links

We introduce the following Corollary 1 proved by Morita *et al.* [MO05].

Corollary 1 (Morita and Ota, 2005). *Suppose that p is a leaf in $\mathsf{T}_{ex}(\boldsymbol{x})$ and its parent node q is an internal node in $\mathsf{T}(\boldsymbol{x})$. Then, $\boldsymbol{w}(p)$ is an MFW of $\boldsymbol{x}$ if and only if node $\sigma(q)$ has two children in $\mathsf{T}(\boldsymbol{x})$.*

Proof. This proof is omitted here (see [MO05]). □

Corollary 1 is derived from Theorem 1 in Section 2.4.1 of this thesis and Corollary 1 can be only applied to a binary string $\boldsymbol{x}$. We generalize

Corollary 1 to any string over finite alphabet as the following Corollary 2.

Corollary 2. *Suppose that p is a leaf in $\mathcal{T}_{\mathsf{ex}}(\boldsymbol{x})$, its parent node q is an internal node except the root in $\mathcal{T}(\boldsymbol{x})$ and $\boldsymbol{w}(q,p)$ is a symbol b. Then, $\boldsymbol{w}(p)$ is an MFW of $\boldsymbol{x}$ if and only if node $\sigma(q)$ has at least two children in $\mathcal{T}(\boldsymbol{x})$ and $b \in \mathcal{L}(\sigma(q))$.*

Proof. Assume that $\boldsymbol{w}(p)$ is an MFW of $\boldsymbol{x}$. The node q has at least one child since q is an internal node, while $b \notin \mathcal{L}(q)$ holds because $\boldsymbol{w}(p)$ is an MFW. Therefore, $\sigma(q)$ has at least two children because $\sigma(p) \in \mathcal{D}(\boldsymbol{x})$ is satisfied from (2.6). Moreover, $\boldsymbol{w}(\sigma(q), \sigma(p))$ is the symbol b since $\boldsymbol{w}(q,p)$ is the symbol b. Hence, $\sigma(q)$ has at least two children and $b \in \mathcal{L}(\sigma(q))$ holds.

Conversely, assume that $\sigma(q)$ has at least two children and $b \in \mathcal{L}(\sigma(q))$. Since $b \in \mathcal{L}(\sigma(q))$ and $\boldsymbol{w}(q,p)$ is the symbol b, $\boldsymbol{w}(\sigma(p)) \in \mathcal{D}(\boldsymbol{x})$ holds. From (2.4), (2.5) and (2.6), $\boldsymbol{w}(p)$ is an MFW of $\boldsymbol{x}$ because $\boldsymbol{w}(p) \notin \mathcal{D}(\boldsymbol{x})$, $\boldsymbol{w}(q) \in \mathcal{D}(\boldsymbol{x})$ and $\boldsymbol{w}(\sigma(p)) \in \mathcal{D}(\boldsymbol{x})$ hold. □

Corollary 2 shows that string $\boldsymbol{w}(p) = a\boldsymbol{w}(\sigma(q))b$ is an element of $\mathcal{A}_I(\boldsymbol{x})$ if and only if $\sigma(q)$ has at least two children in $\mathcal{T}(\boldsymbol{x})$ and $b \in \mathcal{L}(\sigma(q))$, where $a \in \mathcal{X}$. The node $\sigma(q)$ is an internal node in $\mathbb{T}(\boldsymbol{x})$ since $\sigma(q)$ has at least two children in $\mathcal{T}(\boldsymbol{x})$.

For example, Table 4.1 shows the relationship between all MFWs of $\mathcal{A}(\boldsymbol{x})$ for $\boldsymbol{x} = 122123$ and $\mathbb{T}(\boldsymbol{x})$ in Figure 2.2. As shown in (2.8), $\mathcal{A}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$ is given by $\{11, 13, 32, 33, 121, 222, 223, 312, 2122\}$. The node ρ, p_1 and p_2 are internal nodes in $\mathbb{T}(\boldsymbol{x})$. The node q_1 is the shortest leaf q_s of $\mathbb{T}(\boldsymbol{x})$. Table 4.1 shows each element of $\mathcal{A}_I(\boldsymbol{x})$ shown in (2.12) is represented by $a\boldsymbol{w}(p)b$, where p is an internal node in $\mathbb{T}(\boldsymbol{x})$. On the other hand, an

Table 4.1: List of all MFWs of $\mathcal{A}(\boldsymbol{x})$, where $\boldsymbol{x} = 1221231$.

	path-string		MFWs	
node	representation	value	representation	value
ρ	$\boldsymbol{w}(\rho)$	λ	$1\boldsymbol{w}(\rho)1, 1\boldsymbol{w}(\rho)3, 3\boldsymbol{w}(\rho)2, 3\boldsymbol{w}(\rho)3$	$11, 13, 32, 33$
p_1	$\boldsymbol{w}(p_1)$	2	$2\boldsymbol{w}(p_1)1, 2\boldsymbol{w}(p_1)2, 2\boldsymbol{w}(p_1)3$	$121, 222, 223$
p_2	$\boldsymbol{w}(p_2)$	12	$1\boldsymbol{w}(p_2)2$	2122
q_1	$\boldsymbol{w}(q_1)$	3	$\boldsymbol{w}(q_1)2$	32

element of $\mathcal{A}_L(\boldsymbol{x})$ shown in (2.11) is also be represented by $a\boldsymbol{w}(p)b$, while the node p can be an implicit node or an internal node in $\mathbb{T}(\boldsymbol{x})$.

Table 4.1 suggests that it may be sufficient to access all internal nodes and the shortest leaf of $\mathbb{T}(\boldsymbol{x})$ to determine all elements of $\mathcal{A}(\boldsymbol{x})$. Then, the total number of nodes associated with MFWs is at most n.

For an internal node p of $\mathbb{T}(\boldsymbol{x})$, to determine whether a string $a\boldsymbol{w}(p)b$ is an MFW of $\mathcal{A}_I(\boldsymbol{x})$, from (2.4) and (2.6), we need to check if the following two equations (4.1) and (4.2) are satisfied:

$$b \in \mathcal{L}(p) \tag{4.1}$$

$$b \notin \mathcal{L}(q), \tag{4.2}$$

where q satisfies $\boldsymbol{w}(q) = a\boldsymbol{w}(p)$ and $a \in \mathcal{X}$. If (4.1) and (4.2) are satisfied and $q \neq q_s$, then $a\boldsymbol{w}(p)b$ is an MFW of $\mathcal{A}_I(\boldsymbol{x})$.

Note that q given above can be not only an internal node but also either an implicit node or the leaf q_s of $\mathbb{T}(\boldsymbol{x})$ if q in $\mathbb{T}(\boldsymbol{x})$. Moreover, if $q = q_s$, then $\boldsymbol{w}(q_s)b$ belongs to $\mathcal{A}_L(\boldsymbol{x})$.

For each internal node p of $\mathbb{T}(\boldsymbol{x})$, to access to q from p and utilize $\mathcal{L}(q)$

efficiently, we introduce a new pointer called *MF-link* in $\mathbb{T}(\boldsymbol{x})$.

Definition 1 (MF-link). *An* MF-link *is a pointer from an internal node p in $\mathbb{T}(\boldsymbol{x})$ to a node q such that $\boldsymbol{w}(q) = a\boldsymbol{w}(p)$ for $a \in \mathcal{X}$.*

The node q in Definition 1 can be written as a function of p and $a \in \mathcal{X}$ such that

$$q = \gamma_a(p).$$

If q is an internal node of $\mathbb{T}(\boldsymbol{x})$, this function γ_a is the reversed suffix link of Weiner's algorithm [Wei73], called the *link vector* [Gus97]. Any link vector points to only an internal node, while any MF-link points to an internal node, an implicit node or the leaf q_s of $\mathbb{T}(\boldsymbol{x})$. In other words, the set of link vectors is a subset of MF-links.

Since q may be an implicit node, to determine whether $a\boldsymbol{w}(p)c (= \boldsymbol{w}(q)c)$ is an MFW of $\mathcal{A}_I(\boldsymbol{x})$, it is necessary to access to an implicit node from an internal node. It means that link vectors may fail to point some of MFWs of $\mathcal{A}_I(\boldsymbol{x})$ while MF-links can do all of them. Therefore, we use MF-links to obtain all MFWs of $\mathcal{A}_I(\boldsymbol{x})$.

Note that any internal node p of $\mathbb{T}(\boldsymbol{x})$ has at least one MF-link since $\boldsymbol{w}(p)$ appears at least twice in $\boldsymbol{x}$. Figure 4.1 shows MF-links for $\boldsymbol{x} = 1221231$. In Figure 4.1, curved lines represent MF-links in $\mathbb{T}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

The root ρ has three MF-links and p_2 has one MF-link. MF-links do not necessarily point to internal nodes of $\mathbb{T}(\boldsymbol{x})$. They may point to implicit nodes of $\mathcal{T}(\boldsymbol{x})$. Hence, $\gamma_a(p)$ is represented by $(q, \boldsymbol{w})$ where q is an internal node of $\mathbb{T}(\boldsymbol{x})$ that is the nearest ancestor node of $\gamma_a(p)$ in $\mathbb{T}(\boldsymbol{x})$ and $\boldsymbol{w}$ is the string such that $\boldsymbol{w}(\gamma_a(p)) = \boldsymbol{w}(q)\boldsymbol{w}$. Since $\boldsymbol{w}$ is a substring of $\boldsymbol{x}$, say $\boldsymbol{x}[i, j]$,

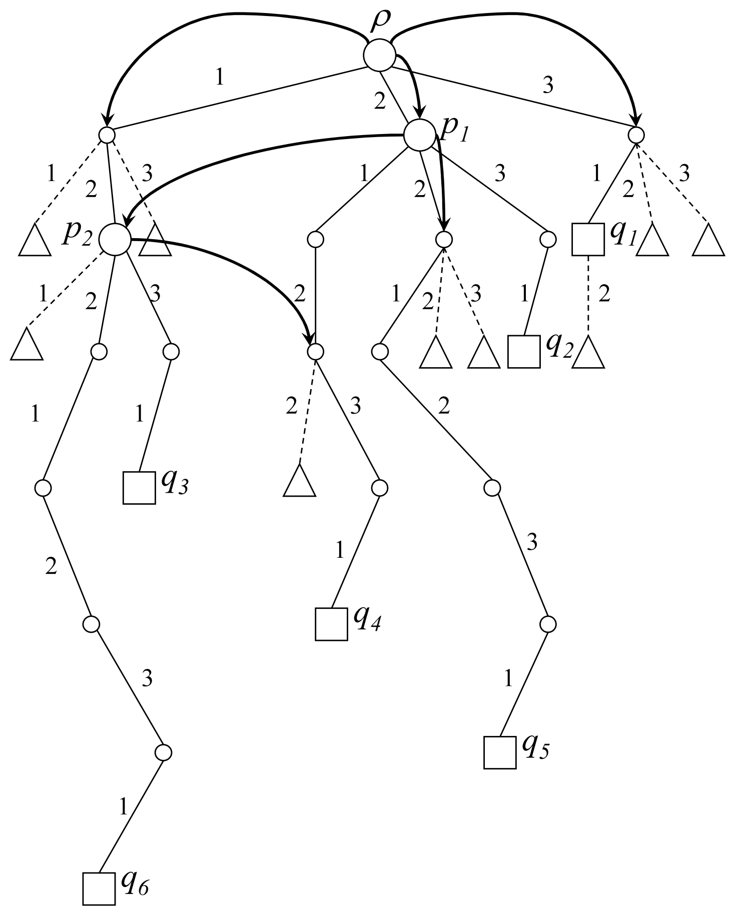


Figure 4.1: Graphical representation of MF-links in $\mathbb{T}(\boldsymbol{x})$, where $\boldsymbol{x}$ = 1221231.

the node $(q, \boldsymbol{w})$ can be written as $(q, [i, j])$. Of course, it may happen that $q = \gamma_a(p)$ and $\boldsymbol{w} = \lambda$. For example, $\gamma_1(\rho) = (\rho, [1, 1])$, $\gamma_2(\rho) = (p_1, \lambda)$ and $\gamma_2(p_2) = (p_1, [4, 5])$ in Figure 4.1, where $[1, 1]$ and $[4, 5]$ are label string 1 and 12, respectively.

To find an MFW of an internal node p, a direct method using a symbol-by-symbol search for $\gamma_a(p)$ travels through $|\boldsymbol{w}(\gamma_a(p))|(= |a\boldsymbol{w}(p)|)$ nodes of $\mathcal{T}(\boldsymbol{x})$. Since this number is proportional to the length of the longest common prefix and it is known that the average length is $O(\log n)$ [DSR92], we can construct all MF-links in $O(n \log n)$ average time.

In the rest of this section, we give a proposition to guarantee that MF-links associated with internal nodes of $\mathbb{T}(\boldsymbol{x})$ can be found in $O(n)$ time. The idea is to do a node-by-node search for $a\boldsymbol{w}(p)$ in $\mathbb{T}(\boldsymbol{x})$ instead of in $\mathcal{T}(\boldsymbol{x})$ to reduce the computational time.

Proposition 1. *Suppose the following conditions hold for an internal node p except the root on $\mathbb{T}(\boldsymbol{x})$ and its parent node $\pi(p)$;*

(1) For the first symbol b of $\boldsymbol{w}(\pi(p), p)$, node $q = (\gamma_a(\pi(p)), b)$ exists on $\mathbb{T}(\boldsymbol{x})$, and

(2) $|\boldsymbol{w}(\gamma_a(\pi(p)), q)| \geq |\boldsymbol{w}(\pi(p), p)|$.

Then $\boldsymbol{w}(\pi(p), p)$ is a prefix of $\boldsymbol{w}(\gamma_a(\pi(p)), q)$, that is, there exists $\boldsymbol{z} \in \mathcal{X}^*$ such that

$$\boldsymbol{w}(\gamma_a(\pi(p)), q) = \boldsymbol{w}(\pi(p), p)\boldsymbol{z}.$$

Proof. Suppose that $\gamma_a(\pi(p))$ exists and that there are $b \in \mathcal{X}$, q satisfying (1) to (2) of Proposition 1. Let $\boldsymbol{v}$ be the label string $\boldsymbol{w}(\pi(p), p)$. We have to prove that there exists $\boldsymbol{t}$ and $\boldsymbol{z}$ such that $\boldsymbol{w}(\gamma_a(\pi(p)), q) = \boldsymbol{tz}$ and $\boldsymbol{t} = \boldsymbol{v}$.

Let $\boldsymbol{w}$ be the longest common prefix between $\boldsymbol{v}$ and $\boldsymbol{t}$. If $|\boldsymbol{w}| < |\boldsymbol{v}|$, $\boldsymbol{w}(\pi(p))\boldsymbol{w}c$ and $\boldsymbol{w}(\pi(p))\boldsymbol{w}d$ are in $\mathcal{D}(\boldsymbol{x})$ where $c, d \in \mathcal{X}$. Thus, there exists an internal node r on $\mathbb{T}(\boldsymbol{x})$ such that $\boldsymbol{w} = \boldsymbol{w}(\pi(p), r)$. However, it contradicts to the construction of $\boldsymbol{v}$. Therefore, $|\boldsymbol{w}| = |\boldsymbol{v}|$. Then it follows that $\boldsymbol{t} = \boldsymbol{v}$. $\square$

By applying Proposition 1, we can find the MF-links of p node-by-node, since $a\boldsymbol{w}(\pi(p))\boldsymbol{w}(\pi(p), p) = \boldsymbol{w}(\gamma_a(p))$, that is, $\gamma_a(p) = (\gamma_a(\pi(p)), \boldsymbol{w}(\pi(p), p))$. To find the MF-links of all internal nodes on $\mathbb{T}(\boldsymbol{x})$, we check if $\gamma_a(\rho)$ exists, and then we examine the internal nodes p in $\mathbb{T}(\boldsymbol{x})$ node-by-node in breadth-first order to determine whether or not a node r with the property $\boldsymbol{w}(r) = a\boldsymbol{w}(p) = \boldsymbol{w}(\gamma_a(\pi(p)))\boldsymbol{w}(\pi(p), p)$ is in $\mathcal{T}(\boldsymbol{x})$. The details of this method will be described next.

4.2.2 Construction Algorithm

First we give a sketch of the algorithm for finding efficiently MF-links on $\mathbb{T}(\boldsymbol{x})$ by using Proposition 1.

Let p be an internal node of $\mathbb{T}(\boldsymbol{x})$ and $\pi(p)$ be the parent of p. Also let $\boldsymbol{v}$ be the label string $\boldsymbol{w}(\pi(p), p)$. Suppose that we are now on node μ of $\mathcal{T}(\boldsymbol{x})$ such that $\boldsymbol{w}(\mu) = a\boldsymbol{w}(\pi(p))$, that is, $\mu = \gamma_a(\pi(p))$. Then we start finding the next node q such that $\boldsymbol{w}(q) = \boldsymbol{w}(\mu)\boldsymbol{v}$ in other words $q = \gamma_a(p)$.

The search process depends on whether μ is an explicit or implicit node.

In the case of an explicit node, we select the child r of μ such that the edge from μ to r is labeled by the string $\boldsymbol{l}$ starting with the first symbol of $\boldsymbol{v}$. From Proposition 1, it follows that $\boldsymbol{l} = \boldsymbol{vz}$ without checking the remaining symbols of $\boldsymbol{v}$. We identify the $|\boldsymbol{v}|$th position on $\boldsymbol{l}$ as $\gamma_a(p)$.

In case of an implicit node, we use the MF-link $(r, \boldsymbol{w})$ of the previous node $\pi(p)$ connected to μ. Let s be the child of μ on $\mathbb{T}(\boldsymbol{x})$. If the first symbol of string $\boldsymbol{t} = \boldsymbol{w}(\mu, s)$ is the same as the first one of $\boldsymbol{v}$, then we can identify the $|\boldsymbol{v}|$th position of $\boldsymbol{t}$ as $\gamma_a(p)$, that is, $(r, \boldsymbol{wv})$.

Hence, in both cases, the MF-links of $\pi(p)$ to p can be located node-by-node. The conditions put forward in Proposition 1 are not satisfied, if node μ has no edge labeled by the string starting with the first symbol of $\boldsymbol{v}$ or if it has an edge with a label that is shorter than $|\boldsymbol{v}|$. In this case it follows that there is no node q in $\mathcal{T}(\boldsymbol{x})$ such that $\boldsymbol{w}(q) = \boldsymbol{w}(\mu)\boldsymbol{v}$.

We now present the algorithm to construct the antidictionary $\mathcal{A}(\boldsymbol{x})$ of $\boldsymbol{x}$ in linear time. We will use a queue Q to store the 3-tuples $(p, \boldsymbol{l}, q)$, where $\boldsymbol{l}$ is the label string $\boldsymbol{w}(p, (p, a))$ and $a \in \mathcal{X}$, and (p, a) is a child of p in $\mathbb{T}(\boldsymbol{x})$. Moreover, we will use the functions Q.push $(\cdot, \cdot, \cdot)$ and Q.pop() to store and retrieve elements from the queue, and Q.is_empty() to check whether the queue is empty. The algorithm uses the procedure construct_suffix_tree($\boldsymbol{x}$) to build the suffix tree $\mathbb{T}(\boldsymbol{x})$ using the algorithm presented in [Ukk95]. This procedure also provides the leaf with the shortest path as a byproduct.

The set $\mathcal{A}(\boldsymbol{x})$ is initially empty. The function add $(\boldsymbol{w})$ adds the MFW $\boldsymbol{w}$ to $\mathcal{A}(\boldsymbol{x})$. The outline of the algorithm is as follows.

```
Algorithm ST2AD
    input  : a string $\boldsymbol{x}$ of length $n$
    output : the antidictionary $\mathcal{A}(\boldsymbol{x})$
begin                                                        1
```

```
/* Step 1: build $\mathbb{T}(\boldsymbol{x})$ and find shortest leaf, $q_s$ */
$(\mathbb{T}(\boldsymbol{x}), q_s) \leftarrow$ construct_suffix_tree($\boldsymbol{x}$);
/* Step 2: add all MFWs of the first class given $q_s$ */
for (each symbol $b \in \mathcal{L}(\rho)$) do
  if (($\sigma(q_s), b$) exists in $\mathcal{T}(\boldsymbol{x})$) then add ($\boldsymbol{w}(q_s)b$)
/* Step 3: add all MFWs of the second class */
if (is_implicit ($\rho$)) then return;
for (each symbol $a \in \mathcal{L}(\rho)$) do begin
  /* initialize */
  $\gamma_a(\rho) \leftarrow \rho$;  $\boldsymbol{v} \leftarrow a$;  Q.push ($\rho, \boldsymbol{v}, \gamma_a(\rho)$));
  while (not(Q.is_empty())) do begin
    /* visit internal nodes in breadth-first order */
    $(p, \boldsymbol{v}, q) \leftarrow$ Q.pop();
    if (($q, \boldsymbol{v}$) exists in $\mathcal{T}(\boldsymbol{x})$ and $(q, \boldsymbol{v}) \neq q_s$) then begin
      /* construct an MF-link */
      $\gamma_a(p) \leftarrow (q, \boldsymbol{v})$
      for (each symbol $c \in \mathcal{L}(p)$) do begin
        if (($\gamma_a(p), c$) not exist in $\mathcal{T}(\boldsymbol{x})$) then
          add ($a\boldsymbol{w}(p)c$);
        if (($p, c$) is an internal node of $\mathbb{T}(\boldsymbol{x})$) then
          Q.push (($(p, c), \boldsymbol{w}(p, (p, c)), \gamma_a(p)$));
      end for;
    end if;
  end while;
end for;
```

```
  return $\mathcal{A}(\boldsymbol{x})$;
end;
```

We first evaluate the time complexity of this algorithm.

Theorem 4. *Given a string $\boldsymbol{x}$ of length n, the* ST2AD($\boldsymbol{x}$) *algorithm can be implemented to run in time $O(n)$.*

Proof. Let the execution time of Step 1, Step 2 and Step 3 of the ST2AD algorithm be S_1, S_2, S_3, respectively. The time complexity $T(n)$ of the proposed algorithm can thus be expressed by $T(n) = S_1 + S_2 + S_3$.

From [Ukk95], we have,

$$S_1 = O(n). \tag{4.3}$$

The parent node π of the shortest leaf q_s of $\mathbb{T}(\boldsymbol{x})$ is created in the last step of the Ukkonen algorithm, and therefore it is ready to use π and $\boldsymbol{w}(\pi, q_s)$ in Step 2.

The cost of one suffix link operation and one traversing node operation is a positive constant, denoted by c_1. The function add $(\boldsymbol{w}(q_s)b)$ in line 6 is performed in constant time also, since the output is represented by a 2-tuple (b, d), where d is the depth of q_s. Therefore, suppose that the number of leaves in $\mathbb{T}(\boldsymbol{x})$ is l, and we have

$$\begin{aligned} S_2 &\leq c_1 \cdot |\boldsymbol{w}(s)| \\ &= c_1(n - l + 1). \end{aligned} \tag{4.4}$$

The time complexity of Step 3 is proportional to the time needed to perform the pattern search operations in line 15 and the cost of a for-loop from line 18 to line 23. It follows from Proposition 1 that the cost of a single search operation is constant, since it is performed node-by-node. With respect to a for-loop, the repetition $|\mathcal{L}(p)| \leq m$ and the alphabet size m is constant, and the function `add` $(a\boldsymbol{w}(p)c)$ in line 20 takes a constant time also, since the output is represented by a 4-tuple (a, c, k, d), where k is the start of the substring $\boldsymbol{w}(p)$ in $\boldsymbol{x}$ and d is the depth of p. Therefore, the maximum cost of a while-loop iteration c_2 is a positive constant.

Since the size of the alphabet $\mathcal{X}$ is m, we have

$$S_3 \leq m \cdot c_2 \xi(\mathbb{T}(\boldsymbol{x})) \tag{4.5}$$

where $\xi(\mathbb{T}(\boldsymbol{x}))$ denotes the number of nodes of $\mathbb{T}(\boldsymbol{x})$. Since $\mathbb{T}(\boldsymbol{x})$ has l leaves, from [SF96], with respect to $\xi(\mathbb{T}(\boldsymbol{x}))$ we have,

$$\frac{l-1}{m-1} \leq \xi(\mathbb{T}(\boldsymbol{x})) \leq \frac{l-1}{2-1} = (l-1). \tag{4.6}$$

Hence, we obtain

$$S_3 \leq m \cdot c_2 (l-1). \tag{4.7}$$

On the other hand, since $a\boldsymbol{w}(p)$ is a prefix of suffixes of $\boldsymbol{x}$ and the number of suffixes of $\boldsymbol{x}$ is n, we have

$$S_3 \leq c_2 \phi(\mathbb{T}(\boldsymbol{x})) \tag{4.8}$$

where $\phi(\mathbb{T}(\boldsymbol{x}))$ denotes the number of edges of $\mathbb{T}(\boldsymbol{x})$.

From (4.6) and [SF96], for fixed l and $m > 1$, we obtain

$$m \cdot \frac{l-1}{m-1} \leq \phi(\mathbb{T}(\boldsymbol{x})) \leq 2 \cdot \frac{l-1}{2-1} = 2(l-1) \tag{4.9}$$

where the left side and the right side denotes in case of all internal nodes have just m edges and two edges, respectively.

Hence (4.7), (4.8) and (4.9), for $m > 1$, we obtain

$$S_3 \leq 2c_2(l-1) \leq m \cdot c_2(l-1). \tag{4.10}$$

Notice that in case of $m = 1$, since ρ is an implicit node, hence $S_3 = 0$.

Let c be the maximum of the two positive constant values c_1 and c_2. From (4.3), (4.4) and (4.10), we have

$$\begin{aligned} T(n) &\leq O(n) + c(n-l+1) + 2c(l-1) \\ &= O(n) + c(n+l-1) \end{aligned} \tag{4.11}$$

Since $1 \leq l \leq n$, it follows that

$$T(n) \leq O(n) + c(2n-1) \tag{4.12}$$

$$= O(n). \tag{4.13}$$

□

To be precise with respect to the time complexity $T(n)$, the cost of a for-loop from line 18 to line 23 is proportional to the size of $\mathcal{L}(p)$. Since $\mathcal{L}(p)$ is a subset of $\mathcal{X}$, the maximum cost of a for-loop iteration is given by km, where m is the size of $\mathcal{X}$ and k is a positive constant. Therefore, the constant cost of c_2 in the proof of Theorem 1 is proportional to m. Hence, since c is $O(m)$ in (4.12), the ST2AD algorithm can be implemented to run in time $O(mn)$.

The following corollary is directly obtained from the proof of Theorem 4.

Corollary 3. *The upper bound on the number of nodes that have to be traversed in the suffix tree $\mathbb{T}(\boldsymbol{x})$ of the string $\boldsymbol{x}$ of length n to obtain all elements of the antidictionary $\mathcal{A}(\boldsymbol{x})$ is given by $2n-1$. It is independent of the size of the alphabet $\mathcal{X}$.*

Proof. From (4.12), the upper bound on the number of traversed nodes of $\mathbb{T}(\boldsymbol{x})$ in Step 2 and Step 3 is given by $2n-1$. □

On the other hand, the MF-Trie to construct an antidictionary using a suffix dawg traverses at most $2n-1$ states since the upper bound on the number of states in a suffix-dawg is $2n-1$. Therefore, the upper bound on the number of traversed nodes using the proposed algorithm is the same as that of the MF-Trie.

The space complexity is determined as follows. In Step 1, the function construct_suffix_tree($\boldsymbol{x}$) requires $O(n)$ space [Ukk95], and the shortest leaf in Step 2 needs a constant space. The number of internal nodes stored in the queue of Step 3 is at most $n-1$. Moreover, we can record MFWs with constant memory by means of the method described in the proof of Theorem 4. Hence, the proposed algorithm constructs the antidictionary

$\mathcal{A}(\boldsymbol{x})$ for a string $\boldsymbol{x}$ of length n in $O(n)$ space. As shown in Table 2.1, the upper bound on the size of an antidictionary of a string is $O(n)$ size.

4.2.3 Experimental Results

To evaluate the performance of the proposed ST2AD algorithm, it was implemented and used to create the antidictionaries for sample strings taken from a memoryless binary information source X with distribution $p = \Pr[X = 0]$.

Table 4.2 shows the number of nodes that were traversed to obtain the antidictionary of a sample string of length up to 5000 with $p = 0.5$ by using the proposed algorithm and the DCA algorithm. Accordingly, we estimate that the proposed algorithm traverses $1.75n$ nodes while the conventional algorithm traverses $0.5n^2$ nodes. The latter estimation coincides with the results reported in [JLS04].

Table 4.2: Number of traversed nodes ($p = 0.5$).

string length	proposed algorithm	S2AD
1000	1747	491454
2000	3498	1981385
3000	5261	4470215
4000	7012	7958209
5000	8775	12446527

We performed more simulations for sample strings taken from a memoryless binary information source with distribution $p = 0.1$ and $p = 0.5$. Figure 4.2 shows the observed relation between the number of nodes that were traversed and the string length and the impact of the distribution p.

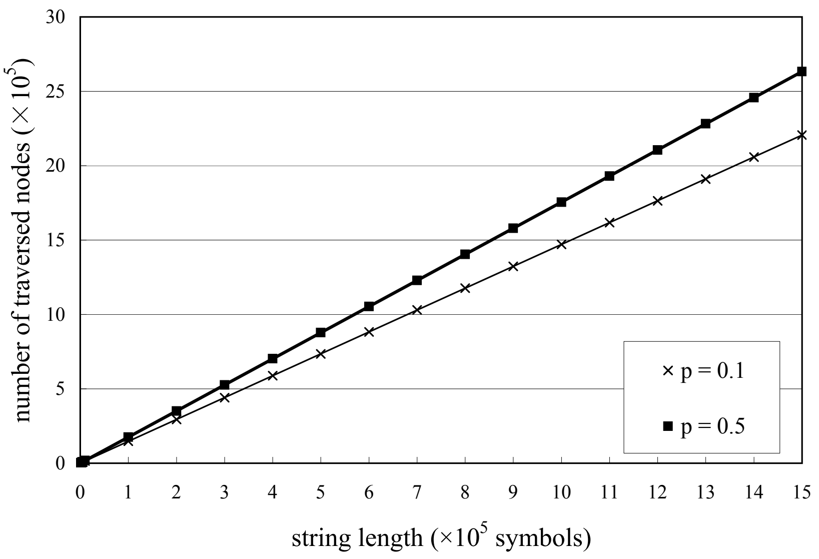


Figure 4.2: Relationship between the string length and the number of traversed nodes ($p = 0.1$ and 0.5) for binary strings.

The number of traversed nodes decreases as p falls since it is proportional to the size of the dictionary.

Then, we used to create the antidictionaries for sample strings taken from a memoryless information source in which every symbol of a finite alphabet $\mathcal{X}$ is generated independently of the other symbols with the same probabilities of symbol generations. We performed simulations for sample strings with the alphabet size 2, 16 and 64, respectively.

Figure 4.3 shows the observed relationship between the number of traversed nodes and the string length for the alphabet size 2, 16 and 64. Accordingly, we estimate that the proposed algorithm traverses $1.75n, 1.30n$ and $1.26n$ nodes with respect to alphabet size 2, 16 and 64, respectively. The results satisfy the upper bound derived from Corollary 3. Notice that from (4.9), the upper bound on the number of traversed nodes is obtained in case of the alphabet being binary. In Figure 4.3, it is shown that the number of traversed nodes decreases, as the alphabet size m grows. The number of traversed nodes depends on the number of edges of $\mathbb{T}(\boldsymbol{x})$, and from (4.8) and from (4.9), the number of edges tends to decrease as m grows.

The computation time needed to complete the ST2AD algorithm on a 3.2 GHz Pentium 4 computer with 2 GB memory is shown in Figure 4.4. For example, it takes about 13 seconds to construct an antidictionary of a random string of length $n = 5 \cdot 10^5$ symbols with the alphabet size 64. This illustrates that the proposed algorithm runs in linear computational time. The computation time is proportional to the alphabet size because the cost of outputting all MFWs in Step 3 is proportional to the alphabet size from Table 2.1.

Figure 4.5 shows the relationship between alphabet size and slopes in computation time graphs with respect to alphabet size 2, 4, 8, 16, 32 and 64. This figure shows the computation time of the ST2AD algorithm is proportional to the alphabet size m for fixed string length. The ST2AD algorithm works in $O(mn)$ time.

4.3 Construction of an AD-trie in linear time

Figure 4.6 shows a scheme of producing $T_{\mathcal{A}}(\boldsymbol{x})$ from $\mathbb{T}(\boldsymbol{x})$ with MF-links. To obtain $T_{\mathcal{A}}(\boldsymbol{x})$, we need to implement the following steps.

Step A : store all elements (MFWs) of $\mathcal{A}(\boldsymbol{x})$ in $\mathbb{T}(\boldsymbol{x})$.

Step B : sort MFWs in each edge of $\mathbb{T}(\boldsymbol{x})$ based on the depth.

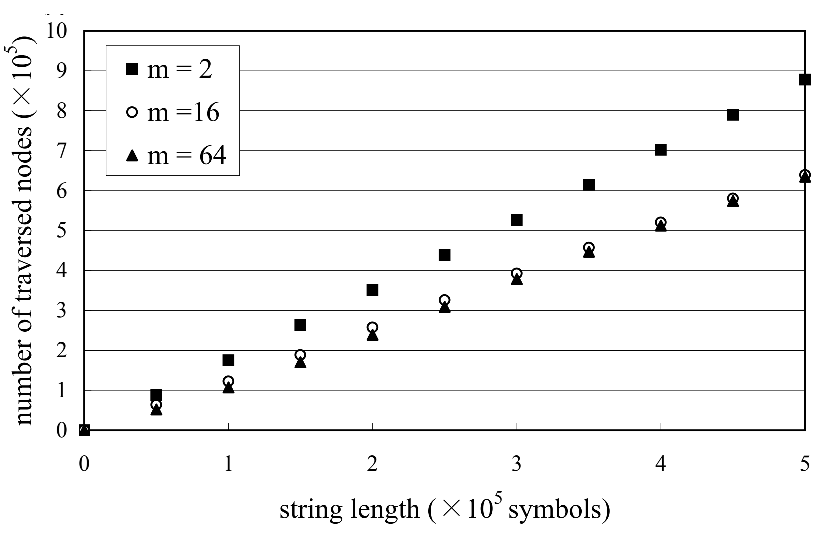


Figure 4.3: Relationship between the string length and the number of traversed nodes ($m = 2, 16$ and 64).

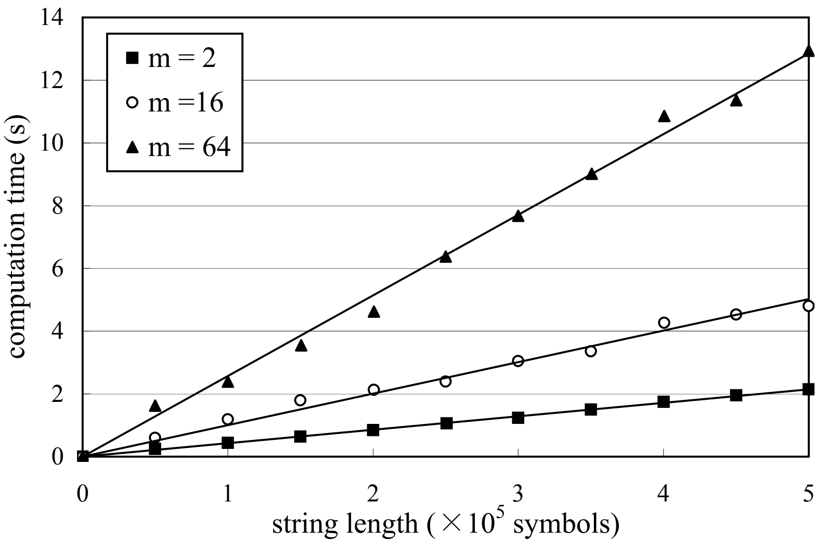


Figure 4.4: Relationship between the string length and the computation time.

Step C : distinguish each edge whether it leads to an AD-node or not.

Step D : copy an edge that leads to an AD-node.

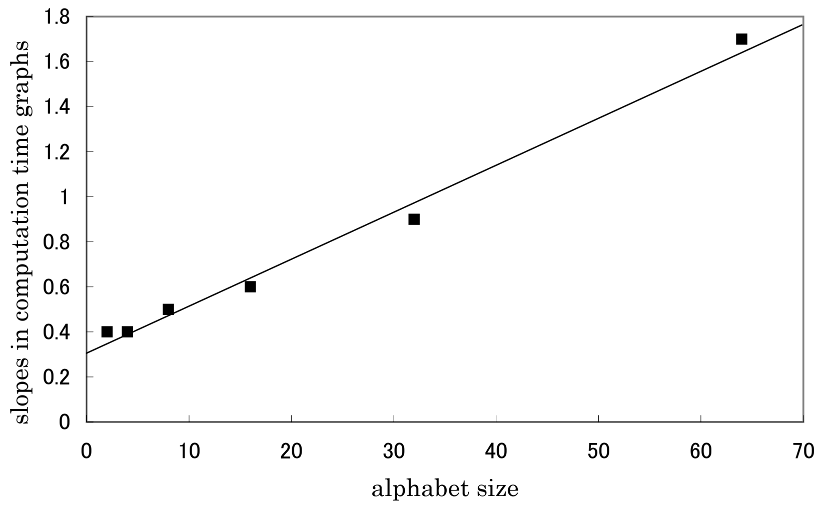


Figure 4.5: Relationship between alphabet size and slopes in computation time graphs.

In Step A, all elements of $\mathcal{A}_I(\boldsymbol{x})$ are stored in internal nodes, and all elements of $\mathcal{A}_L(\boldsymbol{x})$ are stored in the shortest leaf node. MFWs exist in the same edge and they are stored in the same internal node. To obtain $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ by copying edges of $\mathbb{T}(\boldsymbol{x})$ symbol-by-symbol, they need to be sorted based on the depth. Step B sorts MFWs in the same edge based on the depth. Figure 4.7 shows the tree obtained by Step A and B from $\mathbb{T}(\boldsymbol{x})$ with MF-links.

Step C distinguishes each edge whether it leads to an AD-node or not to copy an edge symbol-by-symbol in Step D. Figure 4.8 shows the tree obtained by Step A, B and C. An edge in a solid line is the edge that leads to an AD-node.

Step D copies an edge that leads to an AD-node symbol-by-symbol. Figure 4.9 shows the tree obtained by Step A, B, C and D. The AD-trie $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ can be obtained by Step A, B, C and D from $\mathbb{T}(\boldsymbol{x})$ with MF-links.

In Step A, we need to store $\mathcal{A}(\boldsymbol{x})$ in $\mathbb{T}(\boldsymbol{x})$ with MF-links.

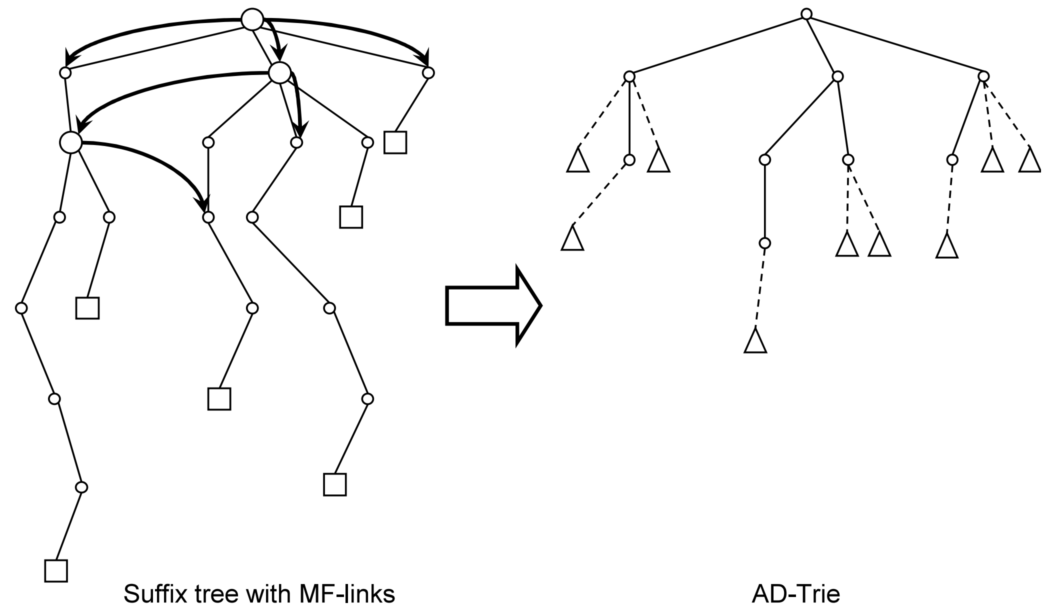


Figure 4.6: A scheme of producing $T_{\mathcal{A}}(\boldsymbol{x})$ from $\mathbb{T}(\boldsymbol{x})$ with MF-links.

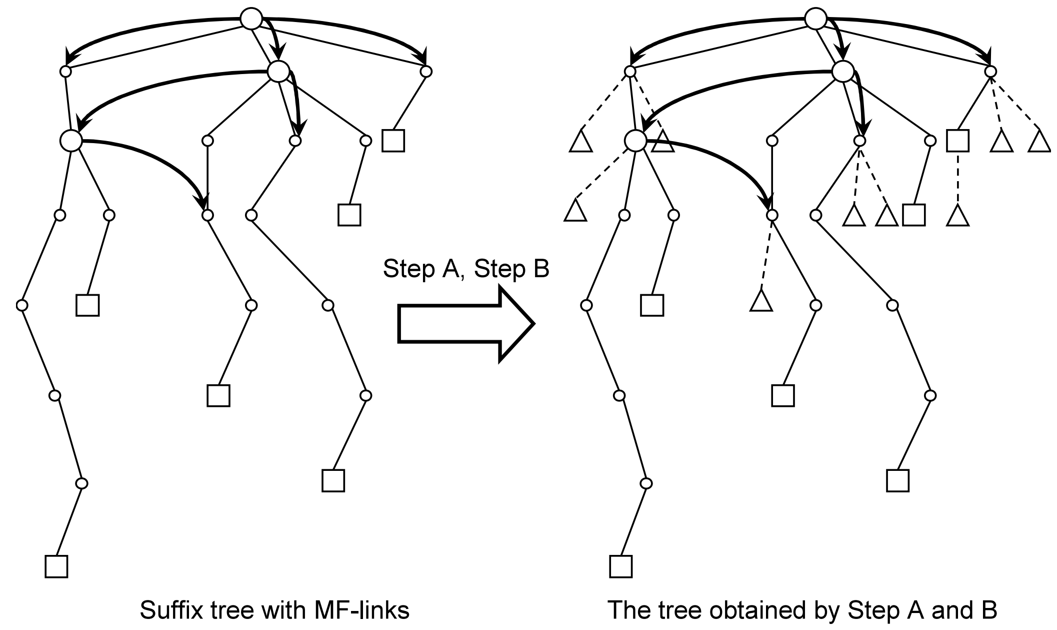


Figure 4.7: The tree obtained by Step A and B.

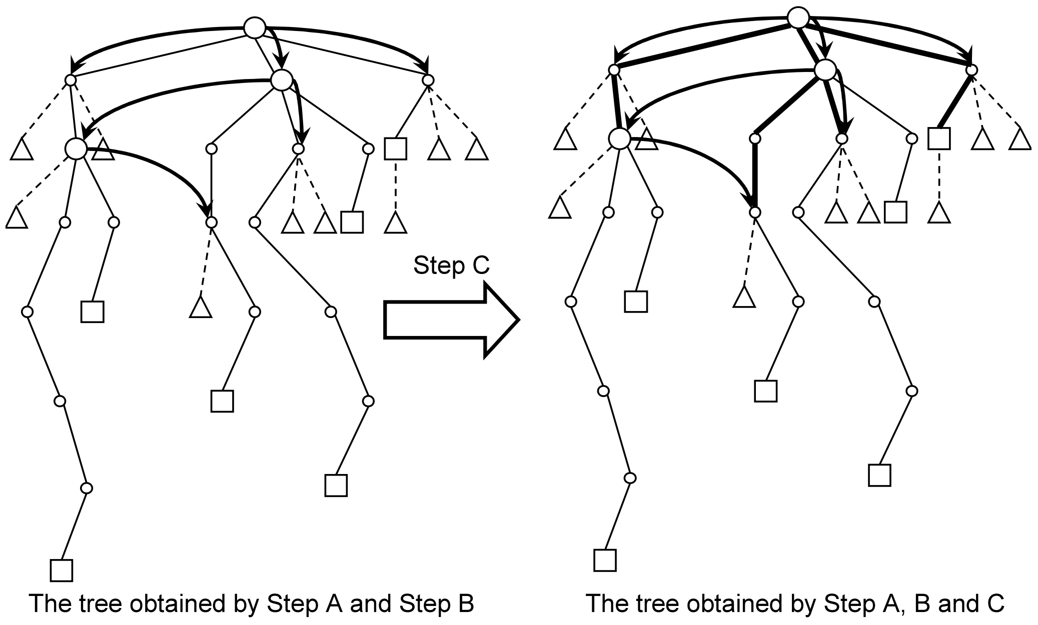

Step C
The tree obtained by Step A and Step B
The tree obtained by Step A, B and C

Figure 4.8: The tree obtained by Step A, B and C.

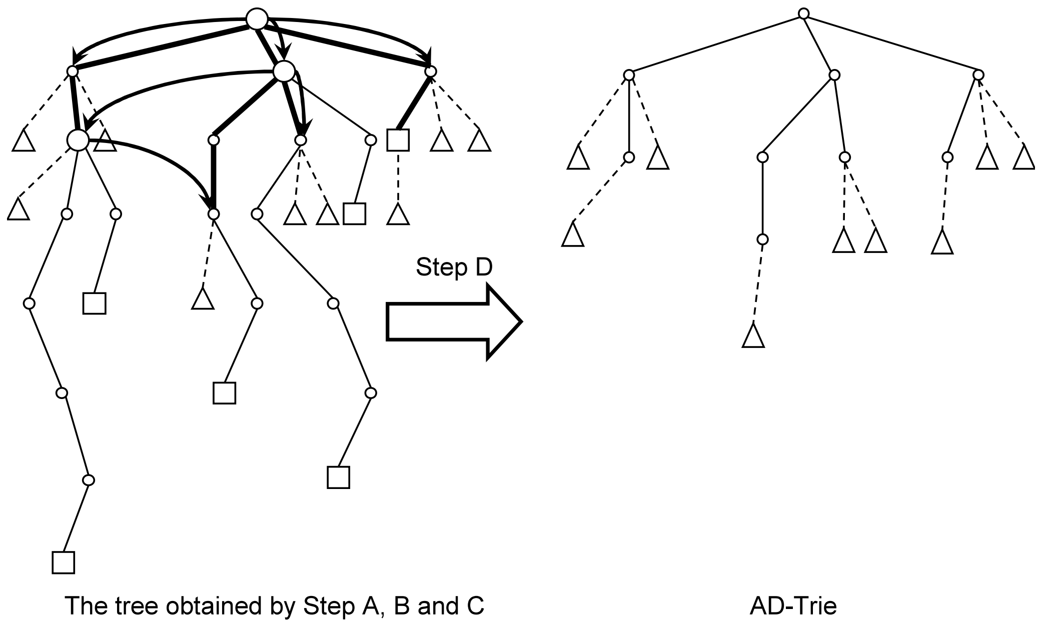


Figure 4.9: The tree obtained by Step A, B, C and D.

First, we consider an element $\boldsymbol{w}(p)c$ of $\mathcal{A}_I(\boldsymbol{x})$. A node $p(=\gamma_a(q))$ is represented by $(r, \boldsymbol{w})$, where r is an internal node of $\mathbb{T}(\boldsymbol{x})$ that the nearest ancestor node of p in $\mathbb{T}(\boldsymbol{x})$ and a string $\boldsymbol{w}$. If $\boldsymbol{w} \neq \lambda$, then $\boldsymbol{w}$ can be represented by a pair of indices $[i, j]$ of string $\boldsymbol{x}$. If the node p is an implicit node of $\mathbb{T}(\boldsymbol{x})$, we can use $(r, [i, j])$ instead of p. Hence, we store a 3-tuples (i, j, c) in node r instead of storing the symbol c in p. If the node p is an internal node of $\mathbb{T}(\boldsymbol{x})$, we store a 3-tuples $(0, 0, c)$ in p.

Next, we consider an element of $\mathcal{A}_L(\boldsymbol{x})$. An element $\boldsymbol{w}$ of $\mathcal{A}_L(\boldsymbol{x})$ is represented by $\boldsymbol{w}(q_s)d$ where q_s is the shortest leaf in $\mathbb{T}(\boldsymbol{x})$ and $d \in \mathcal{X}$. To store an MFW $\boldsymbol{w}$ in $\mathbb{T}(\boldsymbol{x})$, we can use every symbol d such as $\boldsymbol{w}(q_s)d \in \mathcal{A}_L(\boldsymbol{x})$ in the shortest leaf q_s.

Table 4.3 shows representation of an MFW of $\mathcal{A}_I(\boldsymbol{x})$ and of $\mathcal{A}_L(\boldsymbol{x})$ for $\mathbb{T}(\boldsymbol{x})$ with MF-links. For example, Table 4.4 shows the relationship between

Table 4.3: Representation of an MFW of $\mathcal{A}_I(\boldsymbol{x})$ and of $\mathcal{A}_L(\boldsymbol{x})$ for $\mathbb{T}(\boldsymbol{x})$ with MF-links.

class	path-string	representation of MFWs	
$\mathcal{A}_I(\boldsymbol{x})$	$\boldsymbol{w}(\gamma_a(q))c$	(i, j, c) in node r	$\gamma_a(q)$ is implicit node $(r, [i, j])$
$\mathcal{A}_I(\boldsymbol{x})$	$\boldsymbol{w}(\gamma_a(q))c$	$(0, 0, c)$ in node p	$\gamma_a(q)$ is internal node p
$\mathcal{A}_L(\boldsymbol{x})$	$\boldsymbol{w}(q_s)d$	d in node q_s	the shortest leaf q_s.

representation of all MFWs for $\boldsymbol{x} = 1221231$ and $\mathbb{T}(\boldsymbol{x})$ in Figure 4.1.

In Step B, we sort MFWs in the same edge of $\mathbb{T}(\boldsymbol{x})$ based on the depth. The MFWs can be distinguished whether they exists in the same edge or not by an index i stored in an internal node p. To sort them in linear computational time with respect to the string length, we use *radix sort* or *bin sort* algorithm [Sed90, AHU83]. The sorting algorithm uses the array A

Table 4.4: Representations of all MFWs in $\mathbb{T}(\boldsymbol{x})$ with MF-links.

MFW	representation of MFWs	values of MFWs
11	$\boldsymbol{w}((\rho, [1,1]))1$	$(1,1,1)$ in ρ
13	$\boldsymbol{w}((\rho, [1,1]))3$	$(1,1,3)$ in ρ
32	$\boldsymbol{w}((\rho, [6,7]))2$	$(6,7,2)$ in ρ
33	$\boldsymbol{w}((\rho, [6,7]))3$	$(6,7,3)$ in ρ
222	$\boldsymbol{w}((p_1, [3,3]))2$	$(3,3,2)$ in p_1
223	$\boldsymbol{w}((p_1, [3,3]))3$	$(3,3,3)$ in p_1
2122	$\boldsymbol{w}((p_1, [4,5]))2$	$(4,5,2)$ in p_1
121	$\boldsymbol{w}(p_2)1$	$(0,0,1)$ in p_2
312	$\boldsymbol{w}(q_1)2$	2 in q_1

for an edge $\mathbb{T}(\boldsymbol{x})$ whose size m is equal to the depth of the deepest MFW in the edge. For an representation of an MFW (i, j, c) in the node r, this algorithm distributes the symbol c to $A[j-i+1]$ in the array A of the edge between node r to node (r, x_i) in $\mathbb{T}(\boldsymbol{x})$. The deepest MFW of each edge of $\mathbb{T}(\boldsymbol{x})$ is obtained in Step A. If multiple symbols are mapped to identical indices of A, then they are chaining by building a *linked list* [AHU83] of symbols for each array index.

For example, in Figure 4.1, array A_1 of the edge from p_1 to $q_4(= (p_1, 1))$ stores a symbol 2 at $A_1[2]$ instead of an MFW $(4, 5, 2) = 2122$ in Table 4.4. Array A_2 of the edge from ρ to $p_2 = (\rho, 1)$ stores both a symbol 1 and 3 at $A_2[1]$ by using a linked list instead of an MFW $(1, 1, 1) = 11$ and an MFW $(1, 1, 3) = 13$. The size of array A_1 and A_2 is 2 and 1, respectively.

From Theorem 3, the total size of arrays used in the sorting algorithm needs linear size with respect to the string length since $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ has at most $(L_n + 1)n$ nodes, and the sorting algorithm can work in linear time since an

MFW (i, j, c) can be distributed to $A[j-i+1]$ in the array a single pass.

In Step C, we distinguish each edge whether it leads to an AD-node or not. The edge from p to q can be distinguished whether it leads to an AD-node or not by traversing depth-first order in the tree once, because if the subtree, whose q is the root, has the AD-node or the edge has a parent node of the AD-node, then the edge leads to the AD-node.

In Step D, we copy an edge that leads to an AD-node symbol-by-symbol by traversing the tree depth-first order. From Theorem 3, the number of copied nodes in Step D is linear size with respect to the string length.

We now present the algorithm ST2ADT to construct $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ in linear time and space. We will use the same notation used for the ST2AD algorithm. We use the function getIndex $(\gamma_a(p))$ to obtain a 3-tuples (r, i, j) such as $(r, [i, j]) = \gamma_a(p)$ if an MF-link $\gamma_a(p)$ is an implicit node. Moreover, we use the function store_AI $(r, (i, j, c))$ to store a 3-tuples (i, j, c) in an internal node r, where each value is the representation of MFW of $\mathcal{A}_I(\boldsymbol{x})$ in Table 4.3 and the function store_AL (p, d) to store the symbol $d \in \mathcal{X}$ in the shortest leaf p. We use the function set_deepest_MFW (r, i, j) to store the depth of the deepest MFW in the edge between node r and node (r, x_i) in r.

The algorithm uses the procedure set_ADtrie_Edge $(p, (p, a))$ to distinguish edge E between node p to (p, a) whether it is an edge of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ or not. The procedure set_ADtrie_Edge determines that edge E is an edge of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ if the subtree whose (p, a) is the root has an AD-node or the edge E has a parent node of an AD-node by traversing the tree in depth-first order. The algorithm uses also the procedure sort_MFWs $(p, (p, a))$ to sort MFWs on an edge E between node p and (p, a) by using radix sort algorithm based on the depth. For an MFW $\boldsymbol{w}(r,[i,j])b \in \mathcal{A}_I(\boldsymbol{x})$, this procedure distributes a symbol b based on the depth of its MFW to $A[j-i+1]$ in the array $A[1..D]$, where D is the depth of the deepest MFW in edge E.

The outline of the algorithm ST2ADT is as follows.

```
Algorithm ST2ADT
    input  : a string $\boldsymbol{x}$ of length $n$
    output: the AD-trie $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$
begin
  /* Step 1: build $\mathbb{T}(\boldsymbol{x})$ and find shortest leaf $q_s$ */
  $(\mathbb{T}(\boldsymbol{x}), q_s) \leftarrow$ construct_suffix_tree($\boldsymbol{x}$);
  /* Step A: build $\mathbb{T}(\boldsymbol{x})$ with $\mathcal{A}(\boldsymbol{x})$ */
  /* store all MFWs of $\mathcal{A}_L(\boldsymbol{x})$ in $q_s$ */
  for (each symbol $d \in \mathcal{L}(\rho)$) do
    if (($\sigma(q_s), d$) exists in $\mathcal{T}(\boldsymbol{x})$) then store_AL ($q_s, d$)
  if (is_implicit ($\rho$)) then return;
  /* store all MFWs of $\mathcal{A}_I(\boldsymbol{x})$ in internal nodes */
  for (each symbol $a \in \mathcal{L}(\rho)$) do begin
    /* initialize */
    $\gamma_a(\rho) \leftarrow \rho$;  $\boldsymbol{v} \leftarrow a$;  Q.push ($\rho, \boldsymbol{v}, \gamma_a(\rho)$));
    while (not(Q.is_empty())) do begin
      /* visit internal nodes in breadth-first order */
      $(p, \boldsymbol{v}, q) \leftarrow$ Q.pop();
      if (($q, \boldsymbol{v}$) exists in $\mathcal{T}(\boldsymbol{x})$ and $(q, \boldsymbol{v}) \neq q_s$) then begin
        $\gamma_a(p) \leftarrow (q, \boldsymbol{v})$
        **for** (each symbol $c \in \mathcal{L}(p)$) **do begin**
          /∗ *store an MFW of $\mathcal{A}_I(\boldsymbol{x})$ in an internal node* ∗/
          **if** (($\gamma_a(p), c$) not exist in $\mathsf{T}(\boldsymbol{x})$) **then begin**
            **if** ($\gamma_a(p)$ is an implicit node) **then begin**
              $(r, i, j) \leftarrow$ getIndex ($\gamma_a(p)$);
              store_AI ($r, (i, j, c)$) /∗ *store an MFW* ∗/;
              set_deepest_MFW ($r, i, j$); /∗ *the edge from $r$ to $(r, x_i)$* ∗/
            **end if**;
            **else begin** /∗ *an internal node* ∗/
              $r \leftarrow \gamma_a(p)$;
              store_AI ($r, (0, 0, c)$); /∗ *store an MFW* ∗/
            **end else**;
          **end if**;
          **if** (($p, c$) exists in $\mathbb{T}(\boldsymbol{x})$ is an internal node) **then**
            Q.push (($p, c$), $\boldsymbol{w}(p, (p, c))$, $\gamma_a(p)$));
        **end for**;
      **end if**;
    **end while**;
**end for**;
/∗ *Step BC: sort MFWs in each edge and set an edge of $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$* ∗/
**for** (each node $p$ in $\mathbb{T}(\boldsymbol{x})$ in depth-first order) **do begin**
  **for** (each symbol $c \in \mathcal{L}(p)$) **do begin**
    **if** (an MFW exists in the edge between $p$ and $(p, c)$) **then**
      sort_MFWs ($p, (p, c)$); /∗ *sort MFWs in the edge* ∗/
    set_ADtrie_Edge ($p, (p, c)$); /∗ *set an edge of $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$* ∗/
end for;
/* Step D: build $T_{\mathcal{A}}(\boldsymbol{x})$ */
for (each node $p$ in $\mathbb{T}(\boldsymbol{x})$ in depth-first order) do
  copy an edge of $T_{\mathcal{A}}(\boldsymbol{x})$ symbol-by-symbol;
return $T_{\mathcal{A}}(\boldsymbol{x})$;
end;
```

We evaluate the time complexity of this algorithm.

Theorem 5. *Given a string $\boldsymbol{x}$ of length n, the* ST2ADT *algorithm can be implemented to run in time $O(n)$.*

Proof. Let the execution time of Step 1, Step A, Step B, Step C and Step D of the ST2ADT algorithm be S_1, S_A, S_B, S_C, S_D, respectively. The time complexity $T(n)$ of the proposed algorithm can thus be expressed by $T(n) = S_1 + S_A + S_B + S_C + S_D$.

From (4.3), we have

$$S_1 = O(n). \tag{4.14}$$

In Step A, the cost of one operation of getIndex $(\gamma_a(p))$ in line 22 is a positive constant since $\gamma_a(p)$ is represented by $(r, [i, j])$. The cost of one operation of store_AI $(r, (i, j, c))$ in line 23 is a positive constant. Moreover, the cost of one operation of set_deepest_MFW (r, i, j) in line 24 is a positive constant since the depth can be obtained by $j-i+1$. Hence, from Theorem 4, since the cost of the for-loop from line 10 to line 36 is $O(n)$ with respect to the string length n, then we obtain

$$S_A = O(n). \tag{4.15}$$

In Step B and Step C, total computational time of procedure sort_MFWs in line 41 is proportional to total length of arrays allocated for all edges. The total length of arrays is less than or equal to number of nodes of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$. From Theorem 3, the number of nodes of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ is less than or equal $(L_n + 1)n$ nodes. Since total length of arrays is given by $O(n)$ size with respect to the string length n, total computational time of procedure sort_MFWs is given by $O(n)$.

On the other hand, total computational time of procedure set_ADtrie_Edge is proportional to number of nodes of $\mathbb{T}(\boldsymbol{x})$. Since the number of nodes of $\mathbb{T}(\boldsymbol{x})$ is at most $2n$ with respect to the string length n, total computational time of procedure set_ADtrie_Edge is given by $O(n)$. Hence, we obtain

$$S_B + S_C = O(n). \tag{4.16}$$

From Theorem 3, number of copied nodes in Step D is less than or equal $(L_n + 1)n$ nodes. Hence, we obtain

$$S_D = O(n). \tag{4.17}$$

From (4.14), (4.15), (4.16) and (4.17) it follows that

$$T(n) = O(n). \tag{4.18}$$

□

4.4 Conclusion

We proposed an algorithm for construction of an antidictionary of a given string using a suffix tree, and we proved that the time and space computational complexity is linear with the string length. It is shown that the number of traversed nodes to construct an antidictionary is independent of the alphabet size by means of the proposed algorithm, and the upper bound is the same as that of the dawg-based algorithm. Experiments confirm that the proposed algorithm is fast and memory-efficient.

Moreover, we proposed an algorithm to construct the AD-trie for a given string using a suffix tree. We proved that the proposed construction algorithm of the AD-trie works in linear time.

Chapter 5

An On-line DCA with Linear Computational Complexity

5.1 Introduction

The DCA algorithm and its extensions usually work in an off-line manner. On the other hand, an on-line DCA algorithm has the advantage of compression ratios and reading a given string only once. It is reported in [CMRS00] that the on-line DCA algorithm obtained almost the same compression ratios as that of the LZ algorithms for files on Calgary Corpus [Cal], while no further details of their implementations are provided in [CMRS00] or elsewhere in the literature. The straightforward implementation of the on-line DCA algorithm requires the worst case $O(n^2)$ time with respect to the string length n because it needs to update an antidictionary and its encoders whenever a new symbol is read.

In 2006, we proposed a new tree structure called *AD-tree* as an encoder of the DCA algorithm and also proposed the DCA algorithm using an AD-tree instead of an AD-automaton [OM06a]. The proposed algorithm works in linear time with an off-line manner. The AD-tree is a tree structure that stores all proper prefixes of any element of $\mathcal{A}_I(\boldsymbol{x})$ and has reverse MF-links. The AD-tree has properties of both the AD-automaton and the suffix tree. Figure 5.1 shows the relationship among the AD-automaton, the AD-tree and the suffix tree.

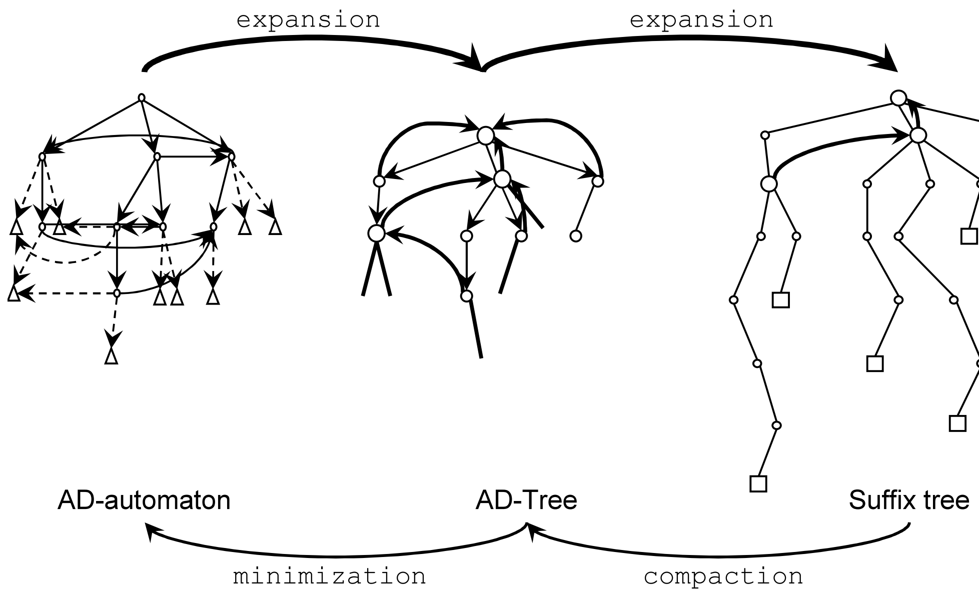


Figure 5.1: The relationship among the AD-automaton, the AD-tree and the suffix tree.

Then, we showed that a suffix tree can be used for the DCA algorithm instead of an AD-tree, and we proposed the DCA algorithm using a suffix tree [OM06d, OM07a]. The proposed algorithm works in linear time with an on-line manner [OM06d, OM07a], and it works by using only the suffix tree without constructing the antidictionary and modifying its encoder.

Moreover, we applied a tree model constructed by the proposed algorithm to a statistical model for an adaptive arithmetic coding, and we also proposed an on-line arithmetic coding using a suffix tree based on antidictionaries [OM06d, OM07a]. We proved that the time complexity of the proposed algorithm is linear with respect to the string length, and we showed that the proposed algorithm achieves better compression ratios for almost all files on Calgary Corpus than those the on-line DCA algorithm [CMRS00]. The average compression ratios of the proposed algorithm were better than that of the on-line DCA and the OHY [OHY05].

This Chapter is organized as follows. Section 5.2 gives the definition of an AD-tree and details the relations between the AD-tree and the AD-automaton. Then, we propose construction algorithm of an AD-tree and the DCA algorithm using the AD-tree. Section 5.3 gives the relations between the AD-tree and the suffix tree. We propose an on-line DCA algorithm using a suffix tree in linear time without constructing an antidictionary and modifying its encoder. Section 5.4 details an on-line arithmetic coding using a suffix tree based on antidictionaries with an on-line manner in linear time. Then, its effectiveness is demonstrated by simulation results. Section 5.5 summarizes our results.

5.2 The DCA algorithm Using an AD-tree

Crochemore *et al.* used an AD-automaton $G_{\mathcal{A}}(\boldsymbol{x})$ to obtain $|\mathcal{V}_i(\boldsymbol{x})|$ and the locus π_i in (3.4). To reduce the computational time for constructing the AD-automaton, we use a subtree of $\mathbb{T}(\boldsymbol{x})$ with reverse MF-links to obtain $|\mathcal{V}_i(\boldsymbol{x})|$

and π_i. In this section, we introduce $\mathbb{T}(\boldsymbol{x})$ with reverse MF-links and show the relationship between $\mathbb{T}(\boldsymbol{x})$ with reverse MF-links and $G_{\mathcal{A}}(\boldsymbol{x})$. Then, we propose the subtree of $\mathbb{T}(\boldsymbol{x})$ with reverse MF-links called *AD-tree* and the DCA algorithm using the AD-tree.

5.2.1 The Relationship between AD-trees and AD-automatons

Let $\mathbb{T}_M(\boldsymbol{x})$ be $\mathbb{T}(\boldsymbol{x})$ with all reverse MF-links for $\boldsymbol{x}$. Figure 5.2 shows $\mathbb{T}_M(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$. To obtain both $|\mathcal{V}_i(\boldsymbol{x})|$ and π_i by using $\mathbb{T}_M(\boldsymbol{x})$, we need answers to two queries:

(a) how to obtain $|\mathcal{V}_i(\boldsymbol{x})|$ for a given π_i using only $\mathbb{T}_M(\boldsymbol{x})$.

(b) how to find the position of π_i in $\mathbb{T}_M(\boldsymbol{x})$.

First, we give an answer to the first query by using the following Lemma 1 and Proposition 2. For any node in $\mathcal{T}^n$, the following Lemma 1 holds.

Lemma 1. *For any node p in $\mathcal{T}^n$, if $a \notin \mathcal{L}_n(p)$ and $a \in \mathcal{L}_n(\rho)$, then there exists the string $\boldsymbol{v}$ such that $\boldsymbol{v} \in \mathcal{S}(\boldsymbol{w}(p)a)$ and $\boldsymbol{v} \in \mathcal{A}(\boldsymbol{x})$.*

Proof. By applying a suffix link $\sigma(\cdot)$ to p for $|\boldsymbol{w}(p)|$ times, we can attain to ρ from p in $\mathcal{T}^n$. Since $a \in \mathcal{L}_n(\rho)$ holds, there exists a node q in $\mathcal{T}^n$ such that $a \notin \mathcal{L}_n(q)$ and $a \in \mathcal{L}_n(\sigma(q))$ on the path of suffix links from p to ρ. Thus, from Theorem 1, $\boldsymbol{w}(q)a \in \mathcal{A}(\boldsymbol{x})$ holds. Moreover, we have $\boldsymbol{w}(q)a \in \mathcal{S}(\boldsymbol{w}(p)a)$ since $\boldsymbol{w}(q) \in \mathcal{S}(\boldsymbol{w}(p))$. Hence, letting the string $\boldsymbol{v}$ be $\boldsymbol{w}(q)a$ completes the proof of Lemma 1. $\square$

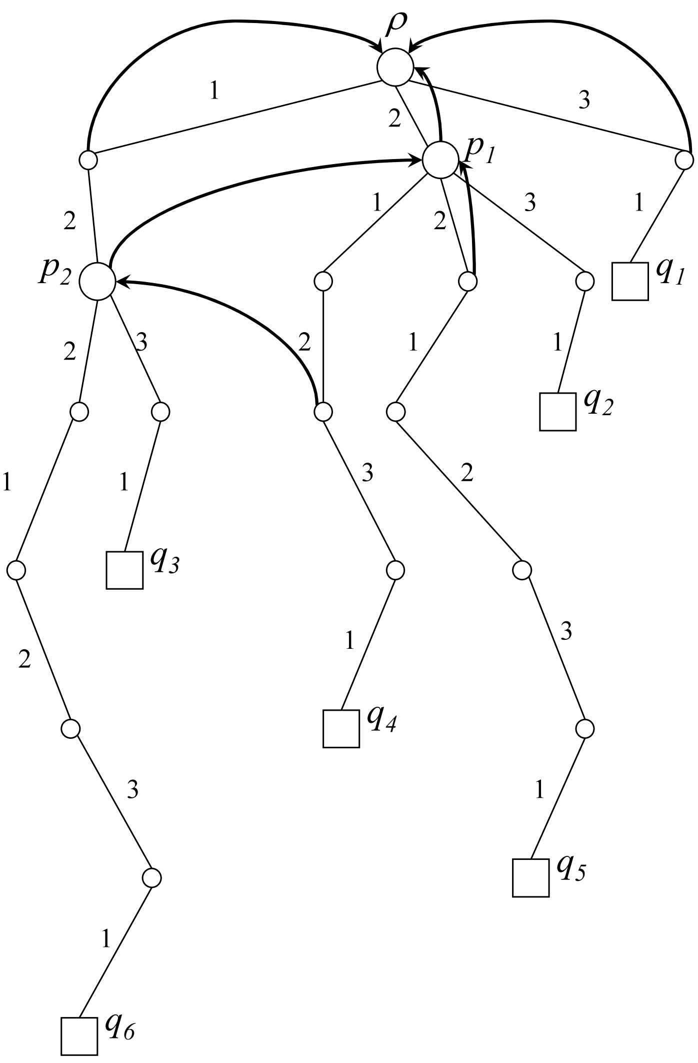


Figure 5.2: Suffix tree with reverse MF-links $\mathbb{T}_M(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

For example, in Figure 2.2, we have $\mathcal{L}_n(p_5) = \{2\}$ and $\mathcal{L}_n(\rho) = \{1, 2, 3\}$. From Lemma 1, there exists the string $\boldsymbol{v}$ such that $\boldsymbol{v} \in \mathcal{S}(\boldsymbol{w}(p_5)3)$ and $\boldsymbol{v} \in \mathcal{A}(\boldsymbol{x})$. It can be verified that the string 13 is the string $\boldsymbol{v}$ because $13 \in \mathcal{S}(213)$ and $13 \in \mathcal{A}(\boldsymbol{x})$ from (2.8). Then, in Figure 2.2, we have $\mathcal{L}_n(p_9) = \{1\}$. From Lemma 1, there exists the string $\boldsymbol{v}$ such that $\boldsymbol{v} \in \mathcal{S}(\boldsymbol{w}(p_9)2)$ and $\boldsymbol{v} \in \mathcal{A}(\boldsymbol{x})$. It can be verified that the string 32 is the string $\boldsymbol{v}$ because $32 \in \mathcal{S}(1232)$ and $32 \in \mathcal{A}(\boldsymbol{x})$ from (2.8).

Then, for any internal node in $\mathcal{T}^n$, the following Proposition 2 holds.

Proposition 2. *For any internal node p in $\mathcal{T}^n$, if $a \notin \mathcal{L}_n(p)$ and $a \in \mathcal{L}_n(\rho)$, then there exists the string $\boldsymbol{v}$ such that $\boldsymbol{v} \in \mathcal{S}(\boldsymbol{w}(p)a)$ and $\boldsymbol{v} \in \mathcal{A}_I(\boldsymbol{x})$.*

Proof. From Lemma 1, there exists the string $\boldsymbol{v}$ such that $\boldsymbol{v} \in \mathcal{S}(\boldsymbol{w}(p)a)$ and $\boldsymbol{v} \in \mathcal{A}(\boldsymbol{x})$. Suppose that $\boldsymbol{v} \in \mathcal{A}_L(\boldsymbol{x})$ holds. Let q_s be the leaf with the shortest path length among all leaves of $\mathcal{T}^n$. Since p is an internal node, we have $\boldsymbol{w}(q_s) \notin \mathcal{S}(\boldsymbol{w}(p))$. Thus we obtain $\boldsymbol{w}(q_s)a \notin \mathcal{S}(\boldsymbol{w}(p)a)$. However, it contradicts to the assumption that $\boldsymbol{v} \in \mathcal{A}_L(\boldsymbol{x})$. Hence, we obtain $\boldsymbol{v} \in \mathcal{A}_I(\boldsymbol{x})$. $\square$

By using Proposition 2, for a given locus π_i, we obtain the equation such that

$$|\mathcal{V}_i(\boldsymbol{x})| = L_n - |\mathcal{L}_n(\pi_i)|. \tag{5.1}$$

From (5.1), we can obtain $|\mathcal{V}_i(\boldsymbol{x})|$ by using the number of children of π_i in $\mathcal{T}^n$. This is an answer to the first query.

For example, we consider the encoding process for $\boldsymbol{x} = 1221231$. As shown in Table 3.1, for $\boldsymbol{x}^5 = 12212$, then π_5 is given by $l(212)$ of $\mathbb{T}_M(\boldsymbol{x})$

in Figure 5.2 since $\boldsymbol{w}_5 = 212$ in (3.3). Since $\mathcal{L}_n(\pi_5) = \{3\}$ in $\mathbb{T}_M(\boldsymbol{x})$ in Figure 5.2, then we have $|\mathcal{V}_5(\boldsymbol{x})| = 2$ from (5.1). It can be verified that $|\mathcal{V}_5(\boldsymbol{x})| = 2$ since both the string 2122 and 121 are MFWs.

Next, we give an answer to the second query. From (3.2), (3.3) and (3.4), π_i exists on a node of $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$ except any AD-node. Let π_0 be ρ of $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$. If π_{i-1} exists on a node of $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$, then from (3.2), (3.3) and (3.4), we have

$$\boldsymbol{w}(\pi_i) \in \{\boldsymbol{w} | \boldsymbol{wv} \in \mathcal{A}_I(\boldsymbol{x}), \boldsymbol{w} \in \mathcal{S}(\boldsymbol{w}(\pi_{i-1})x_i), \boldsymbol{v} \in \mathcal{X}^+\} \cup \{\lambda\}. \qquad (5.2)$$

Let $\mathcal{W}$ be the set of the right-side of (5.2). From (3.3) and (3.4), π_i is the locus of the longest string of $\mathcal{W}$. Therefore, π_i can be obtained by using π_{i-1} and x_i the following scheme.

(i) If $l(\boldsymbol{w}(\pi_{i-1})x_i)$ exists in $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$, then π_i is a π_{i-1}'s child such as (π_{i-1}, x_i) of $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$.

(ii) If $l(\boldsymbol{w}(\pi_{i-1})x_i)$ does not exist in $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$, then π_i is $l(\boldsymbol{t})$ such that $\boldsymbol{t}$ is the longest string of $\mathcal{U} = \{\boldsymbol{u} | \boldsymbol{u} \in \mathcal{S}(\boldsymbol{w}(\pi_{i-1})x_i), l(\boldsymbol{u})$ is a node of $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})\} \cup \{\lambda\}$

From Proposition 2 and (5.1), the latter condition (ii) occurs in case of the following two cases; π_{i-1} is an internal node p such as $|\mathcal{L}(p)| > 1$; π_{i-1} is an implicit node such that it is a destination of an MF-link. Since the string $\boldsymbol{t}$ is an element of $\mathcal{S}(\boldsymbol{w}(\pi_{i-1})x_i)$ and the longest string of $\mathcal{U}$ in (ii), π_i can be obtained by traversing reverse MF-links in $\mathbb{T}_M(\boldsymbol{x})$ starting from π_{i-1} to ρ until $l(\boldsymbol{t})$ is found. This is an answer to the second query.

From the answer of the first query, for a node p in $\mathbb{T}_M(\boldsymbol{x})$, we need $\mathcal{L}_n(p)$ to obtain $|\mathcal{V}_i(\boldsymbol{x})|$. From the answer of the second query, we need a subtree of

$\mathbb{T}_M(\boldsymbol{x})$ that stores all proper prefixes of any element of $\mathcal{A}_I(\boldsymbol{x})$ to obtain π_i. Therefore, to obtain $|\mathcal{V}_i(\boldsymbol{x})|$ and π_i, we need the subtree of $\mathbb{T}_M(\boldsymbol{x})$ that stores all proper prefixes of any element of $\mathcal{A}_I(\boldsymbol{x})$ and $\mathcal{L}_n(p)$ for any node p of that subtree. The resulting subtree is called *AD-tree* and denoted by $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$.

Figure 5.3 shows $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$. Since $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ is the subtree

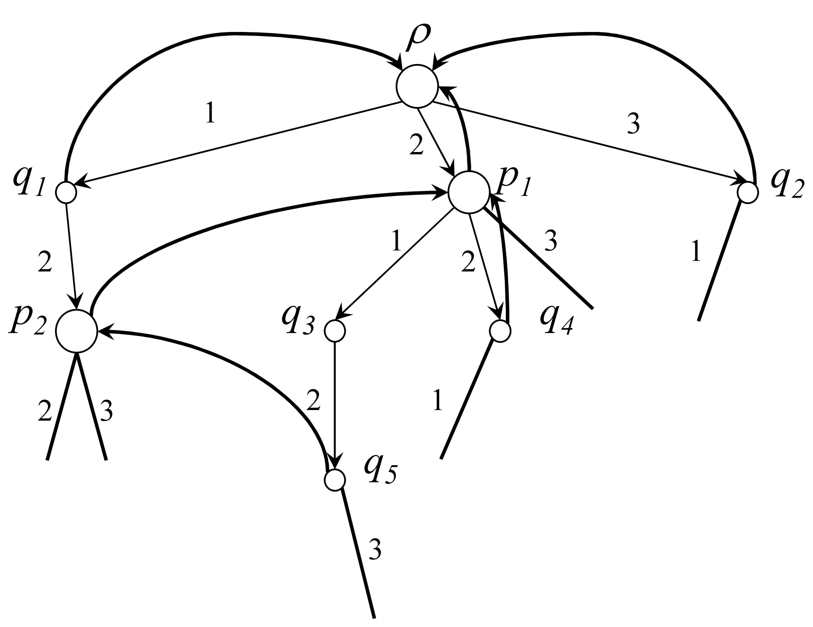


Figure 5.3: AD-tree $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

of $\mathbb{T}_M(\boldsymbol{x})$ and $\mathcal{L}_n(p)$ is defined in $\mathcal{T}(\boldsymbol{x})$, a node exist in $\mathbb{T}_M(\boldsymbol{x})$ but does not exist in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$. In Figure 5.3, an edge labeled symbol a in a solid line connecting from node p to no child is the edge exists in $\mathcal{T}(\boldsymbol{x})$ but the edge does not exist in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$. In Figure 5.3, a curved line represents a reverse MF-link.

By using two answers to the queries (a) and (b), we can use $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ instead of $G_{\mathcal{A}_I}(\boldsymbol{x})$. For example, we show an encoding process of the DCA algorithm using $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$ in Figure 5.3. Starting from ρ, transitions of $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ with $\boldsymbol{x} = 1221231$ will be given as follows:

$$\begin{array}{l} \rho \xrightarrow[1]{1} q_1 \xrightarrow[-]{2} p_2(\xrightarrow{\gamma(\cdot)} p_1) \xrightarrow[2]{2} q_4(\xrightarrow{\gamma(\cdot)} p_1) \xrightarrow[-]{1} \\ q_3 \xrightarrow[-]{2} q_5(\xrightarrow{\gamma(\cdot)} p_2 \xrightarrow{\gamma(\cdot)} p_1 \xrightarrow{\gamma(\cdot)} \rho) \xrightarrow[-]{3} q_2(\xrightarrow{\gamma(\cdot)} \rho) \\ \xrightarrow[-]{1} q_1. \end{array} \quad (5.3)$$

In Eq. (5.3), a symbol of the above of an arrow is an input symbol, while a symbol of lower-side of an arrow is an output symbol. A transition in a bracket denotes the transition by means of a reverse MF-link $\gamma(\cdot)$. No symbol is output in the transition of the reverse MF-link since it corresponds to searching π_i in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$. From Eq. (5.1), since a symbol is outputted when a transition from a node such as $|\mathcal{L}_n(p)| > 1$ occurs, the output occur at internal node ρ and p_2. Therefore, we can obtain the string 12 as a codeword. It is equal to the codeword obtained by using $G_{\mathcal{A}}(\boldsymbol{x})$ in Table 3.2.

Note that we can pass through internal implicit nodes such as q_1 and q_3 since no symbol is outputted. Because an internal implicit node q has only one child, that is, $|\mathcal{L}_n(q)| = 1$.

5.2.2 Construction Algorithm of an AD-tree

An AD-tree can be obtained by using two construction schemes. The first scheme produces the AD-tree from a suffix tree via a suffix tree with reverse MF-links. Figure 5.4 shows the first scheme. This scheme works in linear time with respect to the string length.

The second scheme produces the AD-tree from an AD-trie. Figure 5.5

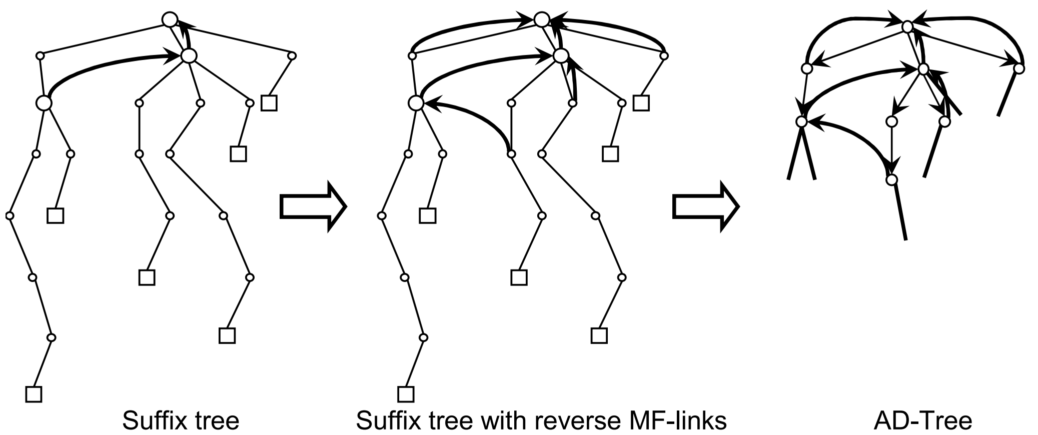


Figure 5.4: A scheme of constructing an AD-tree from a suffix tree.

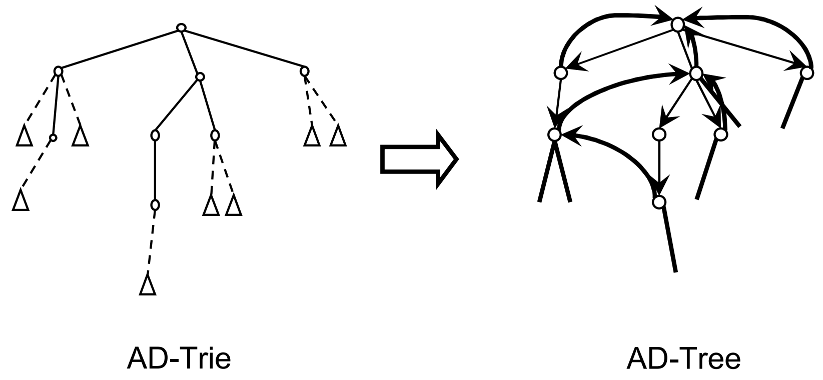


Figure 5.5: A scheme of constructing an AD-tree from an AD-trie.

shows the second scheme. This scheme works in sub-linear computational time. In this section, we propose these two schemes. First, we show the algorithm to construct $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ from $\mathbb{T}(\boldsymbol{x})$ via $\mathbb{T}_M(\boldsymbol{x})$. Since both $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ and $\mathbb{T}_M(\boldsymbol{x})$ have reverse MF-links, we show a representation of a reverse MF-link in these trees. Table 5.1 shows the relationship between the source of a reverse MF-link and the destination of the reverse MF-link in $\mathbb{T}_M(\boldsymbol{x})$. Note that the source represents the starting node of the reverse MF-link

Table 5.1: Relationship between an source of a reverse MF-link and a destination of the reverse MF-link for $\mathcal{A}_I(\boldsymbol{x})$ in the tree $\mathbb{T}_M(\boldsymbol{x})$.

source of reverse MF-link	destination of reverse MF-link
implicit node	internal node
internal node	internal node

Table 5.2: Representation of a reverse MF-link from q to p.

source	destination	reverse MF-link	notes
implicit	internal	(i, j, p) in node r	q is implicit node $(r, [i, j])$
internal	internal	p in node q	use a suffix link of q

and the destination represents the end node of the link. All destinations of reverse MF-links are internal nodes of $\mathbb{T}_M(\boldsymbol{x})$. Let p be a source of a reverse MF-link and let q be a destination of the reverse MF-link. If the node p is an internal node of $\mathbb{T}_M(\boldsymbol{x})$, then we use a suffix link such as $q = \sigma(p)$ instead of the reverse MF-link. If the node p is an implicit node such as $(r, [i, j])$, as shown in Table 4.3, of $\mathbb{T}_M(\boldsymbol{x})$, then we store a pair of indices (i, j) and node q in the node r. To store the reverse MF-link from q to p in $\mathbb{T}_M(\boldsymbol{x})$, we use the similar representation of MFWs in Table 4.3. Table 5.2 shows representation of the reverse MF-link from q to p in $\mathbb{T}_M(\boldsymbol{x})$.

For example, Table 5.3 shows the representation of reverse MF-links of $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$. For example, the reverse MF-link from q_1 to ρ is stored by $(1, 1, \rho)$ in ρ since $q_1 = (\rho, [1, 1])$ is an implicit node, while the reverse MF-link from p_1 to ρ is stored by using suffix link $\sigma(p_1)$ since p_1 is an internal node and $\mathbb{T}_M(\boldsymbol{x})$ stores all suffix links.

Next, we present the Construct AD-Tree algorithm to construct $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$

Table 5.3: An example of reverse MF-links of $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\boldsymbol{x} = 1221231$.

reverse MF-link	value	notes
$q_1 \to \rho$	$(1, 1, \rho)$ in node ρ	$q_1 = (\rho, [1, 1])$
$p_1 \to \rho$	$\sigma(p_1)$	suffix link of p_1
$q_2 \to \rho$	$(6, 6, \rho)$ in node ρ	$q_2 = (\rho, [6, 6])$
$p_2 \to p_1$	$\sigma(p_2)$	suffix link of p_2
$q_4 \to p_1$	$(3, 3, p_1)$ in node p_1	$q_4 = (p_1, [3, 3])$
$q_5 \to p_2$	$(4, 5, p_2)$ in node p_1	$q_5 = (p_1, [4, 5])$

from a given string $\boldsymbol{x}$. The Construct AD-Tree algorithm is similar to the ST2ADT algorithm. The Construct AD-Tree algorithm stores all reverse MF-links in $\mathbb{T}(\boldsymbol{x})$, while the ST2ADT algorithm stores all MFWs in $\mathbb{T}(\boldsymbol{x})$.

We will use the same notation for the ST2ADT algorithm. For a given MF-link $\gamma_a(p)$, we use the function getIndex $(\gamma_a(p))$ to obtain the 3-tuples (r, i, j) such as $(r, [i, j]) = \gamma_a(p)$. We use the function store_RevMFlink $(r, (i, j, p))$ to store (i, j, p) in an internal node r, where the node $(r, [i, j])$ is an source of a reverse MF-link and the node p is a destination of the reverse MF-link. We also use the function set_deepest_RevMFlink (r, i, j) to store the deepest source of reverse MF-link $(r, [i, j])$ on the edge from r to (r, x_i) in internal node r. We can determine whether or not it is the deepest by using value $j - i + 1$.

The function get_deepest_RevMFlink (r, a) to obtain the deepest source of reverse MF-link $(r, [i, j])$ on the edge from r to (r, a). If no source of reverse MF-link exists on the edge from r to (r, a), then the function get_deepest_RevMFlink returns the internal node r.

Let p_d be the deepest source of reverse MF-link on the edge from an internal node p to leaf (p, a) of $\mathbb{T}_M(\boldsymbol{x})$. If no source of reverse MF-link exists on the edge from p to (p, a), then let node p_d be the internal node p. To obtain $\mathbb{T}_M(\boldsymbol{x})$, we use the procedure eliminate_edge $(p_d, (p, a))$ to eliminate an edge from the node p_d to leaf (p, a).

If the edge from p_d to (p, a) is eliminated, then an element of $\mathcal{L}(p_d)$ is lost. From (5.1), since we need $\mathcal{L}(p_d)$ to obtain $|\mathcal{V}_i(\boldsymbol{x})|$, we use the function store (b, p_d) to store the first symbol b of $\boldsymbol{w}(p_d, (p, a))$ in p_d. If p_d is an internal node in $\mathbb{T}(\boldsymbol{x})$, then we store the symbol b in p_d. If $p_d = (r, [i, j])$ is an implicit node in $\mathbb{T}(\boldsymbol{x})$, then we store a 2-tuples (i, b) , where the symbol b is given by x_{j+1}. For example, in Figure 5.3, $\mathcal{L}(p_2) = \{2, 3\}$ in internal node p_2 are stored in p_2, and $\mathcal{L}(q_5) = \{3\}$ in implicit node $q_5 = (p_1, [4, 5])$ is stored by $(4, 3)$ in p_1.

The outline of the algorithm Construct AD-Tree is as follows.

```
Algorithm Construct AD-Tree
    input  : a string $\boldsymbol{x}$ of length $n$
    output : the AD-tree $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$
begin
  /* Step 1: build $\mathbb{T}(\boldsymbol{x})$ */
  $(\mathbb{T}, q_s) \leftarrow$ construct_suffix_tree($\boldsymbol{x}$);
  if (is_implicit ($\rho$)) then return;
  /* Step 2: build $\mathbb{T}_M(\boldsymbol{x})(\boldsymbol{x})$ */
  for (each symbol $a \in \mathcal{L}(\rho)$) do begin
    /* initialize */
    $\gamma_a(\rho) \leftarrow \rho$;  $\boldsymbol{v} \leftarrow a$;  Q.push ($\rho, \boldsymbol{v}, \gamma_a(\rho)$);
    while (not(Q.is_empty())) do begin
```

```
        /* visit internal nodes in breadth-first order */
        $(p, \boldsymbol{v}, q) \leftarrow$ Q.pop();
        if $((q, \boldsymbol{v})$ exists in $\mathsf{T}(\boldsymbol{x}))$ then begin
          $\gamma_a(p) \leftarrow (q, \boldsymbol{v})$
          /* register a reverse MF-link */
          if $(\gamma_a(p)$ points an implicit node) then begin
            $(r, i, j) \leftarrow$ getIndex $(\gamma_a(p))$;
            store_RevMFlink $(r, (i, j, p))$; /* store a reverse MF-link */
            set_deepest_RevMFlink $(r, i, j)$;
          end if;
          for (each symbol $c \in \mathcal{L}(p)$) do begin
            if $((p, c)$ is an internal node of $\mathbb{T}(\boldsymbol{x}))$ then
              Q.push $((p, c), \boldsymbol{w}(p, (p, c)), \gamma_a(p)))$;
          end for;
        end if;
      end while;
    end for;
    /* Step 3: produce $\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x})$ by eliminating edges of $\mathbb{T}_M(\boldsymbol{x})(\boldsymbol{x})$ */
    for (each internal node $p$ of $\mathbb{T}$ in depth-first order) do
      for (each symbol $a \in \mathcal{L}(p)$) do
        if $((p, a)$ is a leaf node) then begin
          $p_d \leftarrow$ get_deepest_RevMFlink $(p, a)$;
          eliminate_edge $(p_d, (p, a))$; /* eliminate an edge $\boldsymbol{w}(p_d, (p, a))$ */
          if $(p_d$ is an internal node) then begin
            store $(a, p_d)$; /* store the first symbol of $\boldsymbol{w}(p_d, (p, a))$ */
        else begin /* an implicit node */
          $(r, i, j) \leftarrow$ getIndex $(p_d)$;
          store $((i, x_{j+1}), r)$; /* store the first symbol of $\boldsymbol{w}(p_d, (p, a))$ */
        end else;
      end if;
    end for;
  end for;
  return $\mathbb{T}(= \mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}))$;
end;
```

We evaluate the time complexity of this algorithm.

Theorem 6. *Given a string $\boldsymbol{x}$ of length n, the* Construct AD-Tree *algorithm can be implemented to run in time $O(n)$.*

Proof. Let the execution time of Step 1, Step 2, Step 3 of the Construct AD-Tree algorithm be S_1, S_2, S_3, respectively. The time complexity $T(n)$ of the proposed algorithm can thus be expressed by $T(n) = S_1 + S_2 + S_3$.

From (4.3), we have

$$S_1 = O(n). \tag{5.4}$$

In Step 2, the function getIndex $(\gamma_a(p))$ in line 16 takes a constant time since $\gamma_a(p)$ is represented by $(r, [i, j])$. The function store_RevMFlink $(r, (i, j, p))$ in line 17 also takes a constant time since this function stores only a 3-tuples (i, j, p) in internal node r. Moreover, the function set_deepest_RevMFlink (r, i, j) in line 18 takes a constant time since the depth can be obtained by $j - i + 1$. Hence, from Theorem 4, since the cost of the for-loop from line 6 to line 26 is $O(n)$ with respect to the string length n, then we obtain

$$S_2 = O(n). \tag{5.5}$$

In Step 3, the cost of one get_deepest_RevMFlink operation is a positive constant since node p_d is stored in internal node p and the alphabet size is a constant. The cost of one eliminate_edge operation is a positive constant since the edge from node p_d to leaf node (p, a) can be represented by a pair of indices of $\boldsymbol{x}$. The cost of one store (b, p) or store $((i, b), p)$ operation is a positive constant since the alphabet size is a constant and the edge represented by a pair of indices of $\boldsymbol{x}$. Moreover, the cost of a for-loop from line 28 to line 41 is $O(n)$ with respect to the string length n since Step 3 traverses the tree in depth-first order and the number of nodes in a suffix tree is at most $2n$. Hence, we obtain

$$S_3 = O(n). \tag{5.6}$$

From (5.4), (5.5) and (5.6), it follows that

$$T(n) = O(n). \tag{5.7}$$

□

We can use the Construct AD-Tree algorithm in the encoding process of the DCA algorithm using an AD-tree, while we *cannot* use the Construct AD-Tree algorithm in the decoding process because this algorithm uses the suffix tree of a given string $\boldsymbol{x}$ to construct the AD-tree. As described in Chapter 3, an encoder of the DCA algorithm needs to send the AD-trie to its decoder.

Therefore, the decoder needs to retrieve an AD-tree from an AD-trie. Then, we propose the algorithm to retrieve the AD-tree from the AD-trie.

In 2005, Morita *et al.* proposed the algorithm, called AD2D, to reproduce $\mathsf{T}(\boldsymbol{x})$ with suffix links from $\mathsf{T}_{\mathcal{A}}(\boldsymbol{x})$ [MO05]. By using the AD2D algorithm, we can retrieve the AD-tree from a given AD-trie.

We propose the AD2ADT algorithm based on the AD2D algorithm to retrieve the AD-tree from a given AD-trie. As described in [MO05], the nodes created by the AD2D algorithm are denoted by *neutral* nodes (see [MO05]). The AD2D algorithm can also retrieve all suffix links of nodes including implicit nodes, therefore we can use a suffix link instead of a reverse MF-link. The AD2D algorithm retrieves redundant nodes for an AD-tree because it reproduces a suffix trie by traversing the AD-trie in breadth-first order. Therefore, we distinguish a reverse MF-link from a suffix link in the AD2ADT algorithm to remove redundant nodes constructed by the AD2D algorithm. We use the function retrieve_RevMFlink (p, q) to retrieve the reverse MF-link from p to q.

The outline of the AD2ADT algorithm is as follows.

```
Algorithm AD2ADT
    input  : an AD-trie $\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x})$
    output : the AD-tree $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$
begin                                                                      1
  /* Step 1:reconstruct a subtree of $\mathsf{T}_I(x)$ with reverse MF-links */     2
  $\mathbb{T} \leftarrow \mathsf{T}_{\mathcal{A}_I}(\boldsymbol{x})$; $\sigma(\rho) \leftarrow \rho$;     3
  for each neutral node $p$ in $\mathbb{T}$ in the breadth-first order do begin     4
```

```
      /* Retrieve nodes of the AD-tree and $\mathcal{L}(p)$ of $\mathbb{T}(\boldsymbol{x})$ */
      create new nodes connecting to $p$ so that $\mathcal{L}(p) = \mathcal{L}(\sigma(p))$;
      /* Retrieve reverse MF-links */
      if ($p$ has an AD-node) then begin
        $q \leftarrow p$;
        while (the reverse MF-link from $q$ to $\sigma(q)$ is not retrieved) do
        begin
          retrieve_RevMFlink $(q, \sigma(q))$;
          $q \leftarrow \sigma(q)$;
        end while;
      if (all parent nodes of AD-nodes are already traversed in $\mathbb{T}$) then
        break;
      end if;
    end for;
    /* Step 2: eliminate redundant nodes of $\mathbb{T}$ */
    for (each node $p$ in $\mathbb{T}$ in depth-first order) do
      eliminate a node from which no path leads to an AD-node or a
        destination of reverse MF-link;
    return $\mathbb{T}(= \mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}))$;
  end.
```

In Step 1, by using the AD2D, a subtree of $T_I(x)$ with reverse MF-links is retrieved. The AD-tree is a subtree of the retrieved tree. Redundant nodes are retrieved in the Step 1 since the AD2D algorithm traverses the tree in breadth-first order until all parent nodes of AD-nodes are traversed. Step 2 eliminates all retrieved redundant nodes to obtain the AD-tree. The AD-tree obtained by the AD2ADT is a trie such that every edge is labelled with a symbol in $\mathcal{X}$.

The AD2ADT algorithm works in a sub-linear computational time because redundant nodes are retrieved in Step 1 and the AD-tree has more than or equal number of nodes of the AD-trie. For example, Figure 5.6 and Figure 5.7 shows $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ and $\mathcal{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\mathcal{A}_I(\boldsymbol{x}) = \{11, 2121\}$ for $\boldsymbol{x} = 12122$, respectively. In Figure 5.6, node p_5 exists in $\mathbb{T}_{\mathcal{A}}(\boldsymbol{x})$, while node p_5 does not exist in $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$.

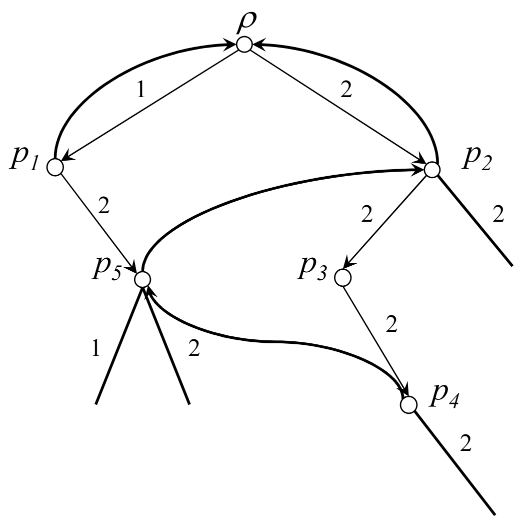


Figure 5.6: AD-tree $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ for $\mathcal{A}_I(\boldsymbol{x}) = \{11, 2121\}$.

On the other hand, the AD2ADT has the practical advantage of reconstructing the AD-tree from an AD-trie of a pruned antidictionary. The AD2ADT can construct the AD-tree from an AD-trie of a given set of MFWs.

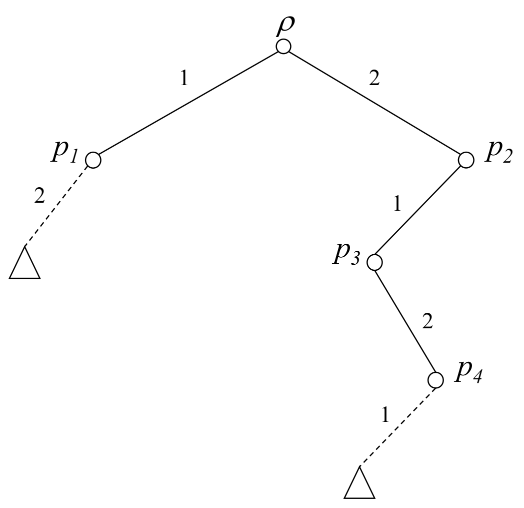


Figure 5.7: AD-trie $T_{\mathcal{A}_I}(\boldsymbol{x})$ for $\mathcal{A}_I(\boldsymbol{x}) = \{11, 2121\}$.

5.2.3 Encoding and Decoding

In this section, we propose the DCA algorithm using an AD-tree. If the AD-tree is constructed by using the Construct AD-Tree, then the DCA algorithm using the AD-tree can traverse nodes with node-by-node, since an edge of the AD-tree is represented by a pair of indices of the string $\boldsymbol{x}$. The algorithm can pass through internal implicit nodes of the AD-tree. On the other hand, if the AD-tree is constructed by using the AD2ADT, then the DCA algorithm traverses nodes with symbol-by-symbol since the AD-tree is a trie such that every edge is labeled with a symbol in $\mathcal{X}$.

In this section, we show that the DCA algorithm using an AD-tree works with symbol-by-symbol. The outline of encoding algorithm DCA Encoder-AT is as follows.

Algorithm DCA Encoder-AT

```
  input  : a string $\boldsymbol{x}$ of length $n$
  output : the 3-tuples (the encoded string $\boldsymbol{\gamma}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{x})$)
begin
  /* Step 1: construct an AD-tree */
  $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ Construct AD-Tree($\boldsymbol{x}$); $T_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ ST2ADT($\boldsymbol{x}$);
  /* Step 2: Encode */
  $p \leftarrow \rho$ of $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$; $i \leftarrow 1$; $\boldsymbol{\gamma} \leftarrow \lambda$;
  for $i := 1$ to $n$ do begin
    /* Output a symbol */
    if $(|\mathcal{L}_n(p)| > 1)$ then
      $\boldsymbol{\gamma} \leftarrow \gamma . x_i$;
    /* traverse reverse MF-links */
    while $((p, x_i)$ not exists in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ or $p \neq \rho)$ do
      $p \leftarrow \sigma(p)$; /* using a reverse MF-link */
    /* traverse an edge */
    if $((p, x_i)$ exists in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}))$ then
      $p \leftarrow (p, x_i)$;
    else
      $p \leftarrow \rho$;
  end for;
  return $(\boldsymbol{\gamma}, n, T_{\mathcal{A}_I}(\boldsymbol{x}))$;
end;
```

We evaluate the time complexity of this algorithm.

Theorem 7. *Given a string $\boldsymbol{x}$ of length n, the* DCA Encoder-AT *algorithm can be implemented to run in time $O(n)$.*

Proof. Let the execution time of Step 1, Step 2 of the DCA Encoder-AT algorithm be S_1 and S_2, respectively. The time complexity $T(n)$ of the algorithm can thus be expressed by $T(n) = S_1 + S_2$.

From Theorem 4 and Theorem 6, we have

$$S_1 = O(n). \tag{5.8}$$

In Step 2, let d_i be the depth of node p in line 10 of phase i of for-loop. The one transition of an edge in line 15 increases the depth by one, while the one transition of a reverse MF-link in line 12 decreases the depth by one. Let c_i be the number of transition of reverse MF-link in phase i, then we have

$$d_{i+1} = d_i - c_i + 1. \tag{5.9}$$

Hence, the total number of transition of reverse MF-link is given by

$$\begin{aligned} \sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} (d_i - d_{i+1} + 1) \\ &= d_1 - d_{n+1} + n. \end{aligned} \tag{5.10}$$

Since the depth d_{n+1} is equal to the depth of p in line 18 of phase n, we have $0 \le d_{n+1} \le n$. Moreover, since d_1 is the depth of ρ, $d_1 = 0$. Hence, from (5.10), we obtain

$$\sum_{i=1}^{n} c_i \le n. \tag{5.11}$$

On the other hand, the number of transition of an edge in line 15 or line 17 of phase i is given by one. Therefore, the total number of down transition of edges c_e is give by

$$c_e = n. \tag{5.12}$$

From (5.11) and (5.12), the total number of transition of nodes in Step 2 is at most $2n$. Therefore, let c be a positive constant, we have

$$\begin{aligned} S_2 &\leq 2cn \qquad (5.13)\\ &= O(n) \qquad (5.14) \end{aligned}$$

From (5.8) and (5.14), it follows that

$$T(n) = O(n). \tag{5.15}$$

□

From (5.13), the DCA Encoder-AT algorithm traverses at most $2n$ nodes in encoding process, while the DCA Encoder-AU traverses surely n states for an arbitrary string. If an AD-tree is constructed by the Construct AD-Tree algorithm, then an edge can be represented by a pair of indices of the string $\boldsymbol{x}$. We can pass through internal implicit nodes of the AD-tree in line 13 of Step 2 of the DCA Encoder-AT.

Next, we show a decoding algorithm of the DCA using the AD-tree. The outline of the decoding algorithm DCA Decoder-AT is as follows.

Algorithm DCA Decoder-AT

```
input  : a 3-tuples (an encoded string $\boldsymbol{\gamma}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{x})$)
output : the string $\boldsymbol{y}(=\boldsymbol{x})$ of length $n$
begin
  /* Step 1: reconstruct an AD-tree */
  $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}) \leftarrow$ AD2ADT($T_{\mathcal{A}_I}(\boldsymbol{x})$);
  /* Step 2: Decode */
  $p \leftarrow \rho$ of $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$; $\boldsymbol{y} \leftarrow \lambda$; $i \leftarrow 1$; $j \leftarrow 1$;
  for $i := 1$ to $n$ do begin
    if $(|\mathcal{L}_n(p)| > 1)$ then
      $a \leftarrow \gamma_j$; $j \leftarrow j + 1$;
    else
      $a \leftarrow b \in \mathcal{L}_n(p)$;
    /* traverse reverse MF-links */
    while $((p, a)$ not exists in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x})$ or $p \neq \rho)$ do
      $p \leftarrow \sigma(p)$;
    /* traverse an edge */
    if $((p, a)$ exists in $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}))$ then
      $p \leftarrow (p, a)$;
    else
      $p \leftarrow \rho$;
  end for;
  return $\boldsymbol{y}(=\boldsymbol{x})$;
end;
```

From Theorem 7, Step 2 takes an linear time with respect to the string length, while Step 1 takes a sub-linear computational time. Therefore, the decoding algorithm of the DCA using an AD-tree works in sub-linear computational time.

5.2.4 Pruning an Antidictionary for an AD-Tree

In this section, we propose a pruning algorithm for an AD-tree without constructing an AD-automaton. The Simple Pruning algorithm uses a value of cost function $c(p)$ such as $\boldsymbol{w}(p) = \boldsymbol{w}(q)a \in \mathcal{A}(\boldsymbol{x})$ to determine whether or not $\boldsymbol{w}(p)$ is eliminated, where p, q is a node of $\mathcal{T}_{\mathcal{A}}(\boldsymbol{x})$ and $a \in \mathcal{X}$. To obtain all values of $c(p)$, the Simple Pruning algorithm requires to construct the AD-automaton and counts frequencies of $\boldsymbol{w}(q)$ in $\boldsymbol{x}$ using the AD-automaton.

On the other hand, we can know the number of $\boldsymbol{w}(q)$ using the subtree of $\mathbb{T}(\boldsymbol{x})$ whose q is the root without the AD-automaton. The number corresponds to the number of leaves of the subtree. The Construct AD-Tree algorithm builds a suffix tree with MF-links in constructing process of an AD-tree, and q is obtained by using MF-links. Hence, we can obtain a value of cost function $c(p)$ by using the suffix tree with MF-links without the AD-automaton. Note that it needs trivial implementation to obtain all value of $c(p)$ since a suffix whose length is less than the path-string of the shortest leaf node of the suffix tree. The pruned AD-trie is obtained by using obtained cost function c and the Simple Pruning algorithm.

5.3 An On-line DCA Using Dynamic Suffix Trees

The off-line DCA algorithms based on an AD-automaton or an AD-tree work with linear time with respect to string length. On the other hand, on-line DCA algorithms based on the AD-automaton or the AD-tree require worst case $O(n^2)$ time with respect to the string of length n because it needs to update an antidictionary and its encoder whenever a new symbol is read.

In Section 5.2, we showed the DCA algorithm using the AD-tree without the AD-automaton. The AD-tree obtained by removing AD-nodes is a subtree of the suffix tree. On the other hand, it is well-known an on-line construction algorithm for suffix trees [Ukk95]. In this section, we show a suffix tree can be used for the encoding of the DCA algorithm instead of the AD-tree, and we propose an on-line DCA algorithm using the dynamic suffix trees without constructing antidictionaries and the encoders.

The implementations of an on-line DCA algorithm require $|\mathcal{V}_i(\boldsymbol{x}^i)|$ to determine whether a symbol x_{i+1} can be eliminated or not. If $|\mathcal{V}_i(\boldsymbol{x}^i)| = L_i - 1$, then x_{i+1} is eliminated.

Let $\boldsymbol{t}_i$ be the longest string in $\mathcal{W}_i(\boldsymbol{x}^i)$, that is,

$$\boldsymbol{t}_i = \xi(\mathcal{W}_i(\boldsymbol{x}^i)). \tag{5.16}$$

Let τ_i be the locus of the longest string $\boldsymbol{t}_i$ in (5.16), that is,

$$\tau_i = l(\boldsymbol{t}_i). \tag{5.17}$$

A direct method obtains $|\mathcal{V}_i(\boldsymbol{x}^i)|$ by using τ_i in $G_{\mathcal{A}_I}(\boldsymbol{x}^i)$ or $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}^i)$. It requires worst case $O(n^2)$ time to obtain τ_i since it needs to update $\mathcal{A}_I(\boldsymbol{x}^i)$ and $G_{\mathcal{A}_I}(\boldsymbol{x}^i)$ or $\mathbb{T}_{\mathcal{A}_I}(\boldsymbol{x}^i)$ whenever a new symbol is read.

To reduce the computational time for updating those data structures, we use only $\mathbb{T}^i$ to obtain $|\mathcal{V}_i(\boldsymbol{x}^i)|$. To obtain $|\mathcal{V}_i(\boldsymbol{x}^i)|$ by using $\mathbb{T}^i$, we need answers to the following two queries:

(a) the condition to eliminate x_{i+1} using τ_i of $\mathbb{T}^i$, and

(b) how to find the position of τ_i in $\mathbb{T}^i$ with an on-line manner in linear time.

We obtain an answer to the first query by using Proposition 2.

If $i > 0$, then τ_i is an internal node in $\mathcal{T}^i$ from (5.17). Thus by using Proposition 2, we obtain

$$|\mathcal{V}_i(\boldsymbol{x}^i)| = L_i - |\mathcal{L}_i(\tau_i)|. \tag{5.18}$$

From (5.18), the condition to eliminate the symbol x_{i+1} is given by $|\mathcal{L}_i(\tau_i)| = 1$. This is an answer to the first query.

Next, we give an answer to the second query. Eq. (5.18) suggests that we need not τ_i but $|\mathcal{L}_i(\tau_i)|$ to eliminate x_{i+1}.

In the Ukkonen algorithm, the locus called *active point* plays a key roll in linear complexity algorithm for the on-line construction of suffix trees. Let α_i be the locus called active point in $\mathbb{T}^i$. The locus α_i is defined as the following.

Definition 2 (active point). *An* active point α_i *in* $\mathbb{T}^i$ *is the locus of string* $\boldsymbol{v}$ *such that the longest string in* $(\mathcal{S}(\boldsymbol{x}^i) \cap \mathcal{D}(\boldsymbol{x}^{i-1}))$*.*

From the definition 2, note that if $i > 0$, then α_i is an internal node in $\mathcal{T}^i$. For the active point α_i, the following Theorem 8 holds.

Theorem 8. *For the active point α_i and the locus τ_i in $\mathcal{T}^i$, if $i > 0$, then*

$$\mathcal{L}_i(\alpha_i) = \mathcal{L}_i(\tau_i)$$

holds.

Proof. From the definition 2, $\boldsymbol{w}(\alpha_i)$ is the longest string such that $\boldsymbol{w}(\alpha_i) \in (\mathcal{S}(\boldsymbol{x}^i) \cap \mathcal{D}(\boldsymbol{x}^{i-1}))$.

Since $i > 0$, τ_i is an internal node in $\mathcal{T}^i$ from (5.17). Thus we have $\boldsymbol{w}(\tau_i) \in \mathcal{D}(\boldsymbol{x}^{i-1})$. From (5.17), we have $\boldsymbol{w}(\tau_i) \in \mathcal{S}(\boldsymbol{x}^i)$. Hence, we obtain $\boldsymbol{w}(\tau_i) \in (\mathcal{S}(\boldsymbol{x}^i) \cap \mathcal{D}(\boldsymbol{x}^{i-1}))$. Clearly $|\boldsymbol{w}(\tau_i)| \leq |\boldsymbol{w}(\alpha_i)|$ holds, thus we obtain $\mathcal{L}_i(\alpha_i) \subseteq \mathcal{L}_i(\tau_i)$.

Suppose that $\mathcal{L}_i(\alpha_i) \subset \mathcal{L}_i(\tau_i)$ holds. A symbol c exists such that $c \notin \mathcal{L}_i(\alpha_i)$ and $c \in \mathcal{L}_i(\tau_i)$. Since $\boldsymbol{w}(\tau_i) \in \mathcal{S}(\boldsymbol{w}(\alpha_i))$ holds, the locus τ_i exists on the path of suffix links from α_i to ρ in $\mathcal{T}^i$. Hence, the node μ exists such that $c \notin \mathcal{L}_i(\mu)$ and $c \in \mathcal{L}_i(\sigma(\mu))$ on the path of suffix links from α_i to τ_i. The node μ is an internal node in $\mathcal{T}^i$ since α_i is an internal node. Hence, the string $\boldsymbol{w}(\mu)c$ satisfies both (2.4) and (2.6), thus we have $\boldsymbol{w}(\mu)c \in \mathcal{A}_I(\boldsymbol{x}^i)$. Moreover, $|\boldsymbol{w}(\tau_i)| < |\boldsymbol{w}(\mu)|$ holds. However, it contradicts to the construction of τ_i since $\boldsymbol{w}(\tau_i)$ is the longest string in $\mathcal{W}_i(\boldsymbol{x}^i)$. Hence, the symbol never exists such as c. Thus $\mathcal{L}_i(\alpha_i) = \mathcal{L}_i(\tau_i)$ holds. □

By using Theorem 8 and from (5.18), we obtain

$$|\mathcal{V}_i(\boldsymbol{x}^i)| = L_i - |\mathcal{L}_i(\alpha_i)|. \tag{5.19}$$

From (5.19), the condition to eliminate x_{i+1} is given by $|\mathcal{L}_i(\alpha_i)| = 1$. In other word, α_i has just one edge in $\mathbb{T}^i$. By using (5.19), we can use α_i instead of τ_i to obtain $|\mathcal{V}_i(\boldsymbol{x}^i)|$. Moreover, by using the Ukkonen algorithm, we can obtain α_i in $\mathbb{T}^i$ with an on-line manner in linear time. This is an answer to the second query.

We present the algorithm to compress a given string based on antidictionaries with an on-line manner in linear time. The algorithm uses the procedure update_suffix_tree$(\cdot,\cdot)$. This procedure constructs $\mathbb{T}^k$ by using the symbol c and the previous tree $\mathbb{T}^{k-1}$. The procedure also provides α_k as a byproducts. Let $\mathbb{T}^0$ be the suffix tree of the null string. The tree $\mathbb{T}^0$ has an only ρ. The outline of the encoding algorithm is as follows.

```
Algorithm On-line DCA Encoder
    input  : a string $\boldsymbol{x}$ of length $n$
    output : the 4-tuples $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, n)$
begin
  /* Step 1: initialize */
  $\boldsymbol{u} \leftarrow \lambda$;   $\boldsymbol{v} \leftarrow x_1$;   $\boldsymbol{w} \leftarrow \lambda$;  $intv \leftarrow 0$;
  $(\alpha_1, \mathbb{T}^1) \leftarrow$ update_suffix_tree$(x_1, \mathbb{T}^0)$;
  for $i := 2$ to $n$ do begin
    /* Step 2: encode */
    if ($x_i \notin \mathcal{L}_{i-1}(\alpha_{i-1})$)
      $\boldsymbol{v} \leftarrow \boldsymbol{v}.x_i$;  $\boldsymbol{w} \leftarrow \boldsymbol{w}.intv$;  $intv \leftarrow 0$;
    else
      $intv \leftarrow intv + 1$;
      if ($|\mathcal{L}_{i-1}(\alpha_{i-1})| > 1$)
        $\boldsymbol{u} \leftarrow \boldsymbol{u}.x_i$;
    /* Step 3: update a suffix tree */
    $(\alpha_i, \mathbb{T}^i) \leftarrow$ update_suffix_tree$(x_i, \mathbb{T}^{i-1})$;
  end for;
  return $(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, n)$;
end;
```

A symbol x_i is eliminated if and only if $|\mathcal{L}_{i-1}(\alpha_{i-1})| = 1$ and $x_i \in \mathcal{L}_{i-1}(\alpha_{i-1})$ hold. The condition $x_i \notin \mathcal{L}_{i-1}(\alpha_{i-1})$ occurs when a new context appeared in the currently suffix tree. Hence, the algorithm also outputs the interval of the condition $x_i \notin \mathcal{L}_{i-1}(\alpha_{i-1})$ and the symbol x_i as the string $\boldsymbol{w}$ and $\boldsymbol{v}$, respectively. In the experiments, we can apply binary representations of integer (cf. [El75]) to the element of $\boldsymbol{w}$. In case of $|\mathcal{X}| = 2$, it is not necessary to output the string $\boldsymbol{v}$ since the symbol of $\boldsymbol{v}$ is predictable. Note that the DCA algorithms can eliminate a symbol if $L_i \geq 2$ holds, while this algorithm can do if $L_i \geq 1$ holds.

Next, the outline of the decoding algorithm is as follows.

Algorithm On-line DCA Decoder

input : a 4-tuple$(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, n)$

```
    output: the string $\boldsymbol{y}(=\boldsymbol{x})$ of length $n$
begin
  /* Step 1: initialize */
  $\boldsymbol{y} \leftarrow v_1$;  $intv \leftarrow 0$;  $j \leftarrow 2$;  $k \leftarrow 1$;
  $(\alpha_1, \mathbb{T}^1) \leftarrow$ update_suffix_tree$(u_1, \mathbb{T}^0)$;
  for $i := 2$ to $n$ do begin
    /* Step 2: decode */
    if $(intv = w_k)$  /* a new context appeared */
      $d \leftarrow v_k$;  $intv \leftarrow 0$;  $k \leftarrow k+1$;
    else
      $intv \leftarrow intv + 1$;
      if $(|\mathcal{L}_{i-1}(\alpha_{i-1})| = 1)$
        $d \leftarrow c \in \mathcal{L}_{i-1}(\alpha_{i-1})$;
      else  /* $|\mathcal{L}_{i-1}(\alpha_{i-1})| > 1$ */
        $d \leftarrow u_j$;  $j \leftarrow j+1$;
    $\boldsymbol{y} \leftarrow \boldsymbol{y}.d$;
    /* Step 3: update a suffix tree */
    $(\alpha_i, \mathbb{T}^i) \leftarrow$ update_suffix_tree$(d, \mathbb{T}^{i-1})$;
  end for;
  return $\boldsymbol{y}$;
end;
```

We can know that a symbol d such that $d \notin \mathcal{L}_{i-1}(\alpha_{i-1})$ will appear if *intv* $= w_k$ holds. In this case, the symbol d is read from the string $\boldsymbol{v}$. Otherwise, the next symbol is an element of $\mathcal{L}_{i-1}(\alpha_{i-1})$. If $|\mathcal{L}_{i-1}(\alpha_{i-1})| > 1$, then we obtain the next symbol from the string $\boldsymbol{u}$.

We evaluate the time complexity of this algorithm.

Theorem 9. *Given a string $\boldsymbol{x}$ of length n, the* On-line DCA Encoder *and the* On-line DCA Decoder *can be implemented to run in time $O(n)$.*

Proof.
Let the execution time of Step 1, Step 2 and Step 3 of the On-line DCA Encoder and the On-line DCA Decoder be $S1, S2, S3$, respectively. The time complexity $T(n)$ of the proposed algorithm can thus be expressed by $T(n) = S1 + (n - 1)(S2 + S3)$.

The cost of operation in line 3 in Step 1 is a positive constant. Hence, from [Ukk95], we have

$$S1 + (n-1)S3 = O(n). \tag{5.20}$$

The locus α_{i-1} is created in Step 1 or Step 3, and therefore it is ready to use α_{i-1} in Step 2.

The cost of Step 2 is proportional to $|\mathcal{L}_{i-1}(\alpha_{i-1})|$. Since $\mathcal{L}_{i-1}(\alpha_{i-1})$ is a subset of $\mathcal{X}$, we have

$$S2 \leq k|\mathcal{X}|, \tag{5.21}$$

where k is a positive constant. From (5.20) and (5.21), we obtain

$$T(n) \leq O(n) + (n-1) \cdot k|\mathcal{X}|.$$

Since $\mathcal{X}$ is finite, the cost $k|\mathcal{X}|$ is a positive constant. Thus, it follows that

$$T(n) = O(n).$$

□

5.4 An On-line Arithmetic Coding Using Dynamic Suffix Trees Based on Antidictionaries

The On-line DCA Encoder constructs a tree structure based on antidictionaries as its encoder. We apply this tree structure to a statistical model for an adaptive arithmetic code. For a internal node p, let $N(c, p)$ be the number of times of a transition from p with symbol c. Each initial value of $N(c, p)$ is assigned by 1 for $c \in \mathcal{L}_{i-1}(p)$. Let $p(d, p)$ be the probability such that $N(d, p)/\sum_{c \in \mathcal{L}_{i-1}(p)} N(c, p)$, where the symbol $d \in \mathcal{L}_{i-1}(p)$. We use $p(d, \alpha_{i-1})$ to encode the symbol d by means of an adaptive arithmetic coding in line 12 of the On-line DCA Encoder. No need to store $N(c, p)$ for an implicit node p because if $|\mathcal{L}_{i-1}(\alpha_{i-1})| = 1$ and $x_i \in \mathcal{L}_{i-1}(\alpha_{i-1})$ hold, then we have $p(x_i, \alpha_{i-1}) = 1$. In this case, the encoder outputs no symbol.

The algorithm uses the procedure AC-E $(c, p, \boldsymbol{s})$ to encode the symbol c by means of cumulative probabilities for an element of $\mathcal{L}_{i-1}(p)$ by an adaptive arithmetic coding order-0, where the string $\boldsymbol{s}$ is a codeword. The algorithm also uses the procedure AC-E $(\cdot)$. This procedure encodes a given string by using an adaptive arithmetic coding order-0.

The outline of the On-line ACDCA Encoder is as follows.

Algorithm On-line ACDCA Encoder

```
    input  : a string $\boldsymbol{x}$ of length $n$
    output : the 4-tuples $(\boldsymbol{\gamma}, \boldsymbol{\kappa}, \boldsymbol{\mu}, n)$
begin
  /* Step 1: initialize */
  $\boldsymbol{\gamma} \leftarrow \lambda$;   $\boldsymbol{v} \leftarrow x_1$;   $\boldsymbol{u} \leftarrow \lambda$;  $intv \leftarrow 0$;
  $(\alpha_1, \mathbb{T}^1) \leftarrow$ update_suffix_tree$(x_1, \mathbb{T}^0)$;
  for $i := 2$ to $n$ do begin
    /* Step 2: encode using tree models */
    if $(x_i \notin \mathcal{L}_{i-1}(\alpha_{i-1}))$
      $\boldsymbol{v} \leftarrow \boldsymbol{v}.x_i$;  $\boldsymbol{w} \leftarrow \boldsymbol{w}.intv$;  $intv \leftarrow 0$;
    else
      $intv \leftarrow intv + 1$;
      if $(|\mathcal{L}_{i-1}(\alpha_{i-1})| > 1)$
        $\boldsymbol{\gamma} \leftarrow$ AC-E $(x_i, \alpha_{i-1}, \boldsymbol{\gamma})$;  $N(x_i, \alpha_{i-1}) \leftarrow N(x_i, \alpha_{i-1}) + 1$;
    /* Step 3: update a suffix tree */
    $(\alpha_i, \mathbb{T}^i) \leftarrow$ update_suffix_tree$(x_i, \mathbb{T}^{i-1})$;
  end for;
  $\boldsymbol{\kappa} \leftarrow$ AC-E $(\boldsymbol{v})$; $\boldsymbol{\mu} \leftarrow$ AC-E $(\boldsymbol{w})$;
  return $(\boldsymbol{\gamma}, \boldsymbol{\kappa}, \boldsymbol{\mu}, n)$;
end;
```

To improve compression ratios, the string $\boldsymbol{v}$ and $\boldsymbol{w}$ are also encoded by an adaptive arithmetic coding in line 16. The string $\boldsymbol{\kappa}$ and $\boldsymbol{\mu}$ is the encoded string $\boldsymbol{v}$ and the encoded $\boldsymbol{w}$ by an adaptive arithmetic coding order-0, respectively.

We consider the computational complexity of the On-line ACDCA Encoder. In line 12, the cost of one operation of AC-E $(x_i, \alpha_{i-1}, \boldsymbol{\gamma})$ is a positive constant since it is proportional to $|\mathcal{L}_{i-1}(\alpha_{i-1})|$ and $|\mathcal{L}_{i-1}(\alpha_{i-1})|$ is a subset of $\mathcal{X}$. Hence, the cost of one operation in line 12 is a positive constant. Then, the cost of one operation in line 16 is a linear with respect to the string length n since the length of $\boldsymbol{v}$ and $\boldsymbol{u}$ is proportional to n. Therefore, from Theorem 9, we have the following Corollary 4.

Corollary 4. *Given a string $\boldsymbol{x}$ of length n, the On-line ACDCA Encoder can be implemented to run in time $O(n)$.*

Next, we show the On-line ACDCA Decoder. The algorithm uses the procedure AC-D $(\boldsymbol{s}, p)$ to decode the symbol c by means of cumulative probabilities of $\mathcal{L}(p)$ and the codeword $\boldsymbol{s}$ by an adaptive arithmetic coding order-0. The algorithm also uses the procedure AC-D $(\cdot)$. This procedure decodes a given string by using an adaptive arithmetic coding order-0. The outline of the ACDCA-D is as follows.

Algorithm On-line ACDCA Decoder

```
    input  : a 4-tuple$(\boldsymbol{\gamma}, \boldsymbol{\kappa}, \boldsymbol{\mu}, n)$
    output : the string $\boldsymbol{y}(= \boldsymbol{x})$ of length $n$
begin                                                                1
  /* Step 1: initialize */                                           2
  $\boldsymbol{v} \leftarrow$ AC-D $(\boldsymbol{\kappa})$;  $\boldsymbol{w} \leftarrow$ AC-D $(\mu)$;
  $\boldsymbol{y} \leftarrow v_1$;  $intv \leftarrow 0$;  $k \leftarrow 1$;
  $(\alpha_1, \mathbb{T}^1) \leftarrow$ update_suffix_tree$(u_1, \mathbb{T}^0)$;
  for $i := 2$ to $n$ do begin
    /* Step 2: decode */
    if $(intv = w_k)$  /* a new context appeared */
      $d \leftarrow v_k$;  $intv \leftarrow 0$;  $k \leftarrow k + 1$;
    else
      $intv \leftarrow intv + 1$;
      if $(|\mathcal{L}_{i-1}(\alpha_{i-1})| = 1)$
        $d \leftarrow c \in \mathcal{L}_{i-1}(\alpha_{i-1})$;
      else  /* $|\mathcal{L}_{i-1}(\alpha_{i-1})| > 1$ */
        $d \leftarrow$ AC-D $(\boldsymbol{\gamma}, \alpha_{i-1})$;  $N(d, \alpha_{i-1}) \leftarrow N(d, \alpha_{i-1}) + 1$;
    $\boldsymbol{y} \leftarrow \boldsymbol{y}.d$;
    /* Step 3: update a suffix tree */
    $(\alpha_i, \mathbb{T}^i) \leftarrow$ update_suffix_tree$(d, \mathbb{T}^{i-1})$;
  end for;
  return $\boldsymbol{y}$;
end;
```

From the argument of the computational time of On-line ACDCA Encoder, the cost of one operation in line 3 is a positive constant, and the cost of one operation in line 15 is a positive constant. Therefore, from Theorem 9, we have the following Corollary 5.

Corollary 5. *Given a string $\boldsymbol{x}$ of length n, the On-line ACDCA Decoder can* *be implemented to run in time $O(n)$.*

5.4.1 Experimental Results

Table 5.4 shows the compression results on Calgary Corpus [Cal] by using the proposed algorithm applied an adaptive arithmetic coding, the on-line DCA algorithm [CMRS00] (DCA), the off-line DCA algorithms applied an adaptive arithmetic coding by Ohkawa *et al.* [OHY05] (OHY), AC using order-2 model [MS98], and gzip [JL02]. Note that Ohkawa *et al.* applied the OHY method to a block of file and determined two parameters such as a block size, a maximum length of MFWs to obtain better compression ratio by preliminary computer simulation for each file.

Table 5.4: Compression results on the Calgary Corpus.

file	proposed	DCA [CMRS00]	OHY	AC order-2	gzip
bib	0.32	0.32	0.30	0.34	0.32
book1	0.41	0.38	0.31	0.37	0.41
book2	0.34	0.35	0.29	0.36	0.34
geo	0.77	0.78	0.68	0.61	0.67
news	0.38	0.43	0.39	0.41	0.38
obj1	0.58	0.61	0.64	0.53	0.48
obj2	0.37	0.45	0.45	0.38	0.33
paper1	0.37	0.40	0.39	0.37	0.35
paper2	0.38	0.39	0.35	0.36	0.36
pic	0.14	0.14	0.10	0.11	0.11
progc	0.37	0.40	0.42	0.37	0.34
progl	0.25	0.28	0.29	0.30	0.23
progp	0.25	0.28	0.31	0.29	0.23
trans	0.21	0.24	0.29	0.30	0.20
Average	0.36	0.39	0.37	0.37	0.34

Table 5.4 shows the proposed algorithm achieved better compression ra-

tios than the DCA for almost all files on Calgary Corpus. The average results give 3% decrease in compressed file size relative to the DCA and 1% decrease in one relative to both the OHY and AC using order-2 model. The proposed algorithm obtained the same compression ratios for four files as gzip and almost the same compression ratios for other files.

5.5 Conclusion

We proposed new data compression algorithms using AD-trees and dynamic suffix trees based on antidictionaries. It is shown that both of them work in linear time with respect to string length. Moreover, the proposed algorithm using the dynamic suffix trees works with linear time in an on-line manner, while the traditional DCA algorithms with an on-line manner require worst case $O(n^2)$ time.

Moreover, we proposed a new on-line arithmetic coding based on antidictionaries using dynamic suffix trees. The proposed algorithm works in linear time, and it was shown that the tree model constructed by the proposed algorithm provides an efficient statistical model for an adaptive arithmetic coding by simulation results.

Chapter 6

An Application of an On-line DCA to Electrocardiogram (ECG)

6.1 Introduction

In recent years, information technology has been used in the biomedical field. This field of research, called biomedical informatics, deals with the resources, devices and methods to optimize acquisition, storage, retrieval and use of biomedical information. In the medical field, it becomes more important to use electronic biomedical data in digital format. Data compression methods are useful for improving the consumption of storage and the amount of transmitting of biomedical information.

The electrocardiogram (ECG) is one of biomedical data. Figure 6.1 shows a schematic representation of normal ECG. As shown in Figure 6.1, a normal

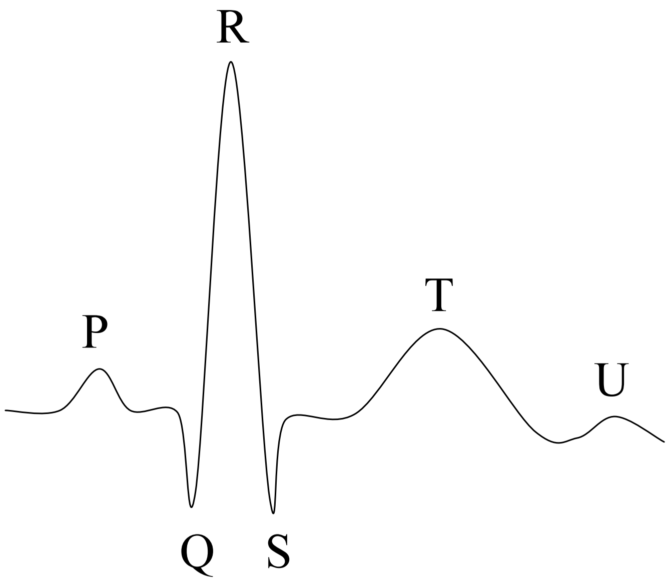


Figure 6.1: A schematic representation of normal ECG.

ECG wave consists of five waves, that is, P, Q, R, S, T and U-wave.

The ECG signal compression is required for two main reasons, effective and economic data storage and on-line transmission of the ECG signals. From the point of view of biomedical data, we need a lossless compression not to lose any significant features of the ECG signals. Moreover, it requires the feasibility of transmitting real-time ECG's over the computer network.

Each ECG signal is an almost periodic waveform and a wave of the ECG differs from other waves with respect to the period and the amplitude. Moreover, a little arrhythmia occurs in the ECG waves. Figure 6.2 shows an example of ECG waves [MIT]. In Figure 6.2, there are seven ECG waves and the third wave from the left is arrhythmia.

These properties of the ECG make it difficult to compress ECG signals by means of lossless data compression, so that numerous lossy data compression

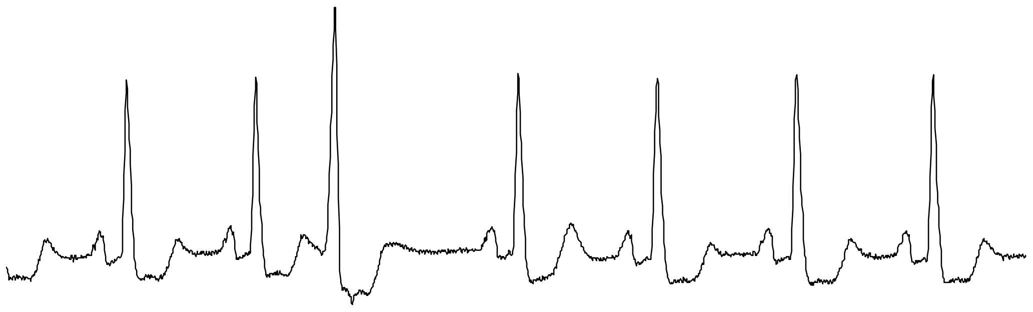


Figure 6.2: An example of ECG waves.

methods for the ECG signals have been proposed [JHSC90, MK93].

To compress the ECG signals with an on-line lossless manner, we can use the On-line DCA Encoder and On-line ACDCA Encoder algorithm, while it is difficult to use these algorithm with respect to memory space since size of long-term monitoring of the ECG signals become extremely large.

In this Chapter, we propose a new on-line lossless data compression for the ECG data using an antidictionary [OM04]. The proposed algorithm constructs the antidictionary by means of the substring of the ECG signals since most of the ECG waves take the similar form of one another. We compress the entire ECG signals by using an AD-automaton of an antidictionary of that substring. Moreover, we study on the length of substring needed to construct the antidictionary whose size is almost same as that of the entire string of ECG using the results of coupon collector's problems [GKP89, Dur99].

Experimental results show that the proposed algorithm gives 10% decrease in compressed file size relative to the LZ algorithm [JL02]. It is shown that the algorithm combined the proposed algorithm with the ACDCA-E gives 15% decrease in compression size relative to the LZ algorithm [JL02]. Moreover, we show the proposed algorithm is available for transmitting real-time by computer simulation.

6.2 An On-line ECG Lossless Compression

We construct an antidictionary by means of the substring of a given binary ECG signals called *learning string* to use the DCA Encoder-AU for binary ECG signals with an on-line manner.

We now show the encoding algorithm ECG-DCA Encoder. The outline of the ECG-DCA Encoder is as follows.

Algorithm ECG-DCA Encoder

```
    input  : a 2-tuples (learning string $\boldsymbol{l}$, ECG signals $\boldsymbol{x}$ of length $n$)
    output : the 4-tuples ($\boldsymbol{\gamma}$, $\boldsymbol{\kappa}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{l})$)
begin
  /* Step 1:Construct an AD-automaton */
  $T_{\mathcal{A}_I}(\boldsymbol{l}) \leftarrow$ ST2ADT($\boldsymbol{l}$);
  $G_{\mathcal{A}_I}(\boldsymbol{l}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{l})$); $intv \leftarrow 0$;
  /* Step 2:Encode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{l})$;
  for $i := 1$ to $n$ do begin
    if (($p, x_i$) is an AD-node) then /* an MFW appears */
      $\boldsymbol{\kappa} \leftarrow \boldsymbol{\kappa}.intv$; $intv \leftarrow 0$; $p \leftarrow \rho$;
    else begin
      if ($p$ has no impossible-transition) then
```

```
        $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma}.x_i$;
      $p \leftarrow (p, x_i)$; $intv \leftarrow intv + 1$;
    **end else**;
  **end for**;
  **return** $(\boldsymbol{\gamma}, \boldsymbol{\kappa}, n, \mathsf{T}_{\mathcal{A}_I}(\boldsymbol{l}))$;
**end**.
```

The string $\boldsymbol{\kappa}$ stores intervals of the occurrence of transition to an AD-node. In the experiments, we applied the binary representation of integer [El75] to an interval. It can occur since we use the AD-automaton of an antidictionary of a learning string such that it is substring of entire ECG signals. If a transition to an AD-node occurs in encoding process, then we reuse the AD-automaton by resetting node p to ρ in line 9.

Then, the outline of the ECG-DCA Decoder is as follows.

```
**Algorithm** ECG-DCA Decoder
    input  : a 4-tuples $(\boldsymbol{\gamma}, \boldsymbol{\kappa}, n, \mathsf{T}_{\mathcal{A}_I}(\boldsymbol{l}))$
    output : the string $\boldsymbol{z}(= \boldsymbol{x})$ of length $n$
**begin**
  /* *Step 1:Reconstruct an AD-automaton* */
  $\mathsf{G}_{\mathcal{A}_I}(\boldsymbol{l}) \leftarrow$ L-Automaton($\mathsf{T}_{\mathcal{A}_I}(\boldsymbol{l})$);
  /* *Step 2:Decode* */
  $p \leftarrow \rho$ of $\mathsf{G}_{\mathcal{A}_I}(\boldsymbol{l})$; $\boldsymbol{z} \leftarrow \lambda$; $j \leftarrow 1$; $k \leftarrow 1$; $intv \leftarrow 0$;
  **for** $i := 1$ **to** $n$ **do begin**
    **if** $(intv = \kappa_k)$ **then** /* *an MFW appears* */
```

```
      $z_i \leftarrow a \notin \mathcal{L}(p)$; $p \leftarrow \rho$; $k \leftarrow k+1$;
    **else begin**
      **if** ($p$ has no impossible-transition) **then begin**
        $z_i \leftarrow \gamma_j$; $j \leftarrow j+1$;
      **else**
        $z_i \leftarrow a \in \mathcal{L}(p)$;
      $p \leftarrow (p, z_i)$; $intv \leftarrow intv+1$;
    **end else**;
  **end for**;
  **return** $\boldsymbol{z}(=\boldsymbol{x})$;
**end**.
```

We consider the computational complexity of the ECG-DCA Encoder and the ECG-DCA Decoder. From Theorem 3 and Theorem 4, the cost of Step 1 of the ECG-DCA Encoder and the ECG-DCA Decoder are linear with respect to the string length. The cost of Step 2 of the ECG-DCA Encoder and the ECG-DCA Decoder is linear since the total number of traversed nodes is n. Therefore, both the ECG-DCA Encoder and the ECG-DCA Decoder can work in linear time and space.

Next, we show the algorithm combined the proposed algorithm with an adaptive arithmetic coding. We will use the same notation of the ACDCA algorithm. The outline of the ECG-ACDCA Encoder is as follows.

Algorithm ECG-ACDCA Encoder

```
  input : a 2-tuples (learning string $\boldsymbol{l}$, ECG signals $\boldsymbol{x}$ of length $n$)
```

```
    output: the 4-tuples ($\boldsymbol{\gamma}$, $\boldsymbol{\kappa}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{l})$)
begin
  /* Step 1:Construct the AD-automaton */
  $T_{\mathcal{A}_I}(\boldsymbol{l}) \leftarrow$ ST2ADT($\boldsymbol{l}$);
  $G_{\mathcal{A}_I}(\boldsymbol{l}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{l})$); $intv \leftarrow 0$; $\boldsymbol{\gamma} \leftarrow \lambda$;
  /* Step 2:Encode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{l})$;
  for $i := 1$ to $n$ do begin
    if (($p, x_i$) is an AD-node) then /* an MFW appears */
      $\boldsymbol{\kappa} \leftarrow \boldsymbol{\kappa}.intv$; $intv \leftarrow 0$; $p \leftarrow \rho$;
    else  begin
      if ($p$ has no impossible-transition) then begin
        $\boldsymbol{\gamma} \leftarrow$ AC-E ($x_i, p, \boldsymbol{\gamma}$); /* an adaptive arithmetic coding */
        $N(x_i, p) \leftarrow N(x_i, p) + 1$;
      end if;
      $p \leftarrow (p, x_i)$; $intv \leftarrow intv + 1$;
    end else;
  end for;
  return ($\boldsymbol{\gamma}$, $\boldsymbol{\kappa}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{l})$);
end.
```

Next, the outline of the ECG-ACDCA Decoder is as follows. We will use the same notation of the ACDCA algorithm.

Algorithm ECG-ACDCA Decoder

```
    input  : a 4-tuples ($\boldsymbol{\gamma}$, $\boldsymbol{\kappa}$, $n$, $T_{\mathcal{A}_I}(\boldsymbol{l})$)
    output : the string $\boldsymbol{z}(=\boldsymbol{x})$ of length $n$
begin
  /* Step 1:Reconstruct the AD-automaton */
  $G_{\mathcal{A}_I}(\boldsymbol{l}) \leftarrow$ L-Automaton($T_{\mathcal{A}_I}(\boldsymbol{l})$);
  /* Step 2:Decode */
  $p \leftarrow \rho$ of $G_{\mathcal{A}_I}(\boldsymbol{l})$; $\boldsymbol{z} \leftarrow \lambda$; $j \leftarrow 1$; $k \leftarrow 1$; $intv \leftarrow 0$;
  for $i := 1$ to $n$ do begin
    if ($intv = \kappa_k$) then /* an MFW appears */
      $z_i \leftarrow a \notin \mathcal{L}(p)$; $p \leftarrow \rho$; $k \leftarrow k+1$;
    else begin
      if ($p$ has no impossible-transition) then begin
        $z_i \leftarrow$ AC-D ($\gamma, p$);
        $N(z_i, p) \leftarrow N(z_i, p) + 1$;
      end if;
      else
        $z_i \leftarrow a \in \mathcal{L}(p)$;
      $p \leftarrow (p, z_i)$; $intv \leftarrow intv + 1$;
    end else;
  end for;
  return $\boldsymbol{z}(=\boldsymbol{x})$;
end.
```

From the computational complexity of the ECG-DCA Encoder and the ECG-DCA Decoder, both the ECG-ACDCA Encoder and the ECG-ACDCA Decoder can also work in linear time and space with respect to the string length.

6.3 Estimation of the Length of Learning String

An antidictionary for the ECG-DCA Encoder is produced from a suffix tree of a given learning string. From Remark 2 [CMR98], there is a one-to-one correspondence between the dictionary and the antidictionary. Hence, it needs the expected length of the learning string $\mathbf{E}[L]$ whose dictionary of the learning string stores an element of all normal ECG signals of a whole ECG data.

We subdivide ECG data into peak-to-peak intervals by using a peak of the R-wave, and each interval is normalized to the fixed N symbols. Figure 6.3 shows the normalized ECG wave.

Let $t_k[i]$ be the amplitude of i-th symbol of k-th intervals, where $1 \leq i \leq N$. Let $T[i]$ be the set of $\{t_k[i] \mid 1 \leq k \leq I\}$, where I is the number of intervals. Since an ECG wave differs from other waves with respect to the amplitude, for a fixed i, there exists a range such as $\mathsf{min}(T[i]) \leq t_k[i] \leq \mathsf{max}(T[i])$. We use the function $\mathsf{min}(\cdot)$ and $\mathsf{max}(\cdot)$ to obtain the minimum and the maximum value for a given set of values, respectively. In the observation of ECG signals, for a fixed i and for an arbitrary k, the behavior of value $t_k[i]$ looks like a random variable in the range $[\mathsf{min}(T[i]), \mathsf{max}(T[i])]$.

Therefore, we assume that the set of values $\mathsf{min}(T[i]), ..., \mathsf{max}(T[i])$ has a discrete uniform distribution. If the values are distributed uniformly,

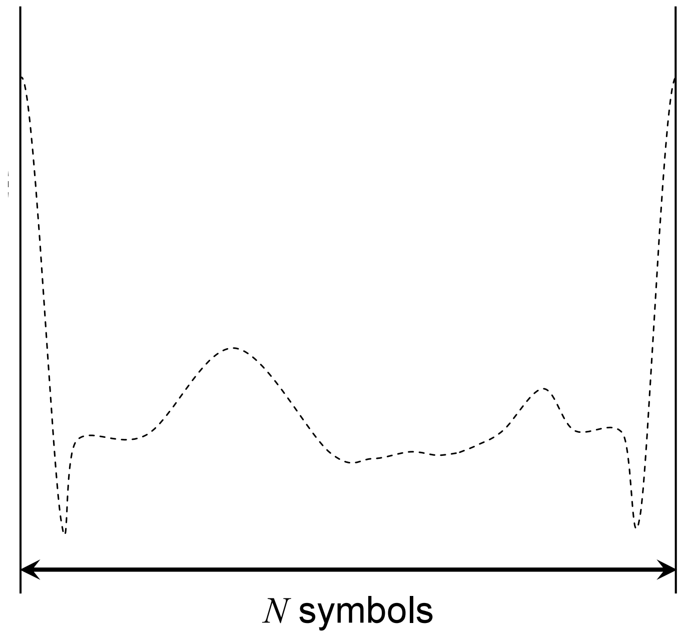


Figure 6.3: The normalized ECG wave.

then we can obtain the expected waiting symbols to appear all values of $[\mathsf{min}(T[i]), ..., \mathsf{max}(T[i])]$ by applying Coupon Collector's Problem [GKP89, Dur99]. In the coupon collector's problem, if each box at a product contains on of a set of M coupons, how many boxes do you need to buy before getting all the coupons? If the coupons are distributed uniformly, then the expected number of boxes to be bought is given by

$$MH_M, \tag{6.1}$$

where H_M is the M-th harmonic number such that $H_M = 1 + \frac{1}{2} + ... + \frac{1}{M}$.

Let d be the value such as $(\mathsf{max}(T[i]) - \mathsf{min}(T[i]) + 1)$. The expected waiting symbols to appear all values of $[\mathsf{min}(T[i]), ..., \mathsf{max}(T[i])]$ is given by

$$NdH_d, \tag{6.2}$$

where N is the period of interval as shown in Figure 6.3. From [GKP89], the approximation of Eq. (6.1) is given by

$$M \log_e M, \tag{6.3}$$

where the constant value e denotes the base of natural logarithm.

From (6.3) and (6.2), the approximation of Eq. (6.2) is given by

$$Nd \log_e d. \tag{6.4}$$

We investigated the dynamic range of normal ECG wave by means of ECG data of MIT-BIH Arrhythmia Database [MIT]. The ECG data is digitized at 360 samples per second per channel with 11-bit resolution. We subdivide the entire ECG into peak-to-peak intervals by using a peak of the R-wave, and each interval is normalized to the fixed N symbols. In our preliminary experiments, N was fixed to 288. Then, in our experiments for 385 normalized intervals ($I = 385$), we obtained the dynamic range d as follows.

$$20 \leq d \leq 55. \tag{6.5}$$

Hence, from (6.4), (6.5) and $N = 288$, the expected length of leaning string is given by

$$17256 \leq \mathbf{E}[L] \leq 63476. \tag{6.6}$$

6.4 Experimental Results

To evaluate the performance of the proposed ECG-DCA and ECG-ACDCA algorithm, both of them were implemented to compress twenty-three ECG files [MIT]. The ECG signals used in the experiment have been sampled at a rate of 360 samples/s. Each sample is represented as an 16-bit code by padded five zero of one sample with 11-bit resolution. The size of each ECG file has 650,000 samples (about 30 min.). From Eq. (6.6), we used 50,000 samples taken from the prefix of each ECG files as the length of a learning string for the ECG-DCA and the ECG-ACDCA algorithm. The maximum restricted length of an MFW of an antidictionary is denoted by R. We used 2 symbols (32-bit) as the length R.

Table 6.1 shows the compression results on the ECG files by using gzip [JL02], the DCA algorithm, the proposed the ECG-DCA and the ECG-ACDCA algorithm. The DCA algorithm works with an off-line manner, while gzip and the proposed algorithms work with an on-line manner. Experimental results show the average result of the ECG-DCA and the ECG-ACDCA give 10% and 15% decrease in compressed file size relative to gzip, respectively. The average result of the DCA algorithm and the ECG-DCA for the restricted length of an MFW to 32-bit shows the proposed ECG-DCA achieved 1% better compression ratio than the DCA algorithm. In our experiment on a 3.2 GHz Pentium 4 with 2 GB memory, it took about 7 second (1.5Mbit/s) to finish encoding one

ECG file in the ECG-DCA and it took about 822 second (13kbit/s) to finish encoding one ECG file in the ECG-ACDCA algorithm. Hence, both of the proposed algorithm can implement to compress the ECG files for real-time since the sampling bit-rate of ECG is 5.7kbit/s.

We performed more simulations for the learning string taken from the ECG file of length up to a whole size (10.4Mbit). Figure 6.4 shows the relationship between length of learning string and compression ratios. In Figure 6.4, the value $d = 20$ and $d = 50$ denotes the lower length and upper length of the value d in Eq. (6.6) for three ECG files. Experimental results show that the length of the learning strings to obtain better compression ratio exists in the range of our estimation of Eq. (6.6).

6.5 Conclusion

We proposed a new on-line lossless data compression for the ECG data using antidictionaries. The proposed algorithm constructs an antidictionary by means of the substring of the ECG signals using the properties such that most of the ECG waves take the similar form of one another.

Experimental results showed the proposed algorithm gives 15% decrease in compressed file size relative to the popular compress algorithms such as the LZ algorithm. We showed the proposed algorithm is available of transmitting real-time by computer simulation.

Moreover, we studied on the length of substring needed to construct an antidictionary whose size is almost the same as that of the entire string of ECG using the results of coupon collector's problems. Experimental results

Table 6.1: Compression results for the ECG files.

ECG	gzip	DCA R=32-bit	ECG-DCA R=32-bit	ECG-DCA	ECG-ACDCA
100	0.386	0.391	0.332	0.313	0.255
101	0.415	0.367	0.360	0.347	0.269
102	0.413	0.344	0.334	0.319	0.262
103	0.430	0.375	0.352	0.339	0.285
104	0.447	0.394	0.379	0.365	0.292
105	0.474	0.381	0.381	0.372	0.335
106	0.482	0.403	0.382	0.376	0.324
107	0.588	0.415	0.441	0.438	0.400
200	0.492	0.413	0.392	0.383	0.329
201	0.373	0.343	0.337	0.318	0.259
202	0.432	0.348	0.336	0.333	0.298
203	0.537	0.421	0.432	0.429	0.385
205	0.368	0.376	0.347	0.321	0.256
207	0.467	0.366	0.357	0.353	0.308
208	0.509	0.398	0.387	0.384	0.345
209	0.472	0.394	0.384	0.379	0.329
210	0.430	0.369	0.331	0.325	0.277
212	0.505	0.404	0.397	0.394	0.354
213	0.554	0.411	0.449	0.446	0.399
214	0.489	0.371	0.370	0.366	0.318
215	0.489	0.413	0.395	0.386	0.331
217	0.545	0.394	0.406	0.412	0.360
219	0.476	0.366	0.406	0.401	0.345
Average	0.469	0.386	0.378	0.369	0.318

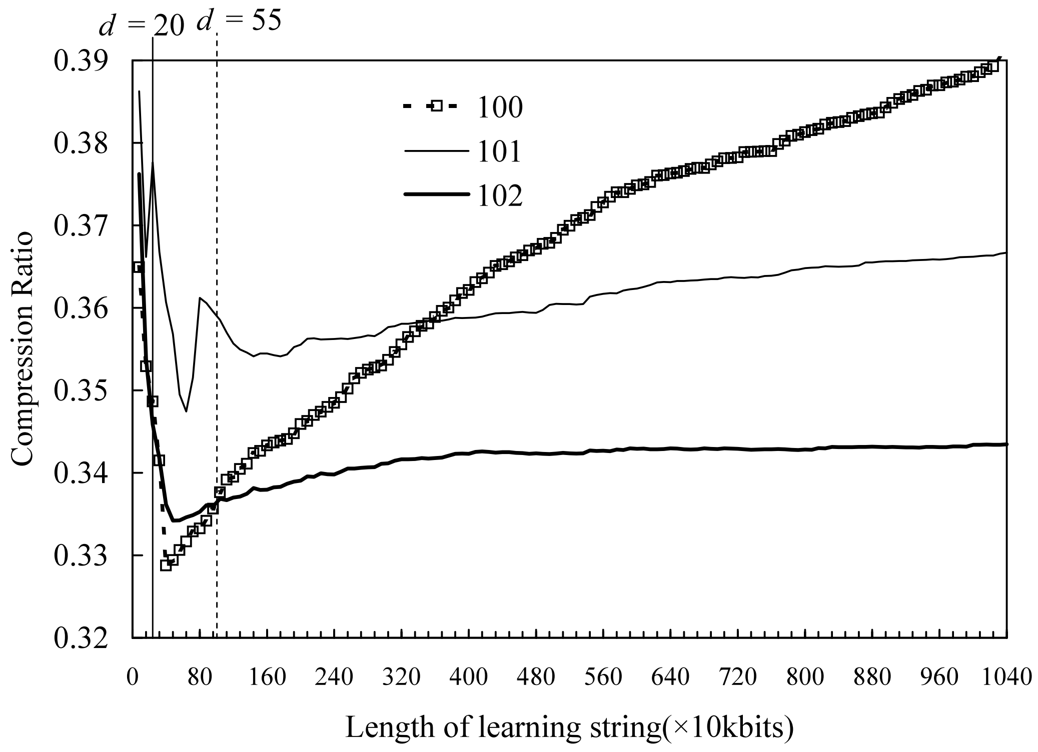


Figure 6.4: Relationship between compression ratios and length of learning strings.

show its validity of estimation.

Chapter 7

Conclusion

We have focused on the construction algorithms of an antidictionary using a tree structure in linear time and an on-line lossless statistical data compression by means of a tree model based on a dictionary and an antidictionary in linear time.

In Chapter 2 and Chapter 3, we generalized the traditional algorithms for only binary strings to any string over finite alphabet. In Chapter 2, we proposed the construction algorithm of an antidictionary of any string over finite alphabet using a suffix trie. In Chapter 3, we proposed the DCA algorithm using an AD-automaton with an adaptive arithmetic coding for any string over finite alphabet.

In Chapter 4, we proposed the construction algorithm of an antidictionary using a suffix tree. We proved that the time and space complexity is linear with the string length. It was shown that the number of traversed nodes to construct an antidictionary is independent of the alphabet size by means of the proposed algorithm. It is the same upper bound, that is $2n - 1$

nodes, as that of the traditional construction algorithm using a suffix dawg. Moreover, experimental results confirmed that the proposed algorithm is fast and memory-efficient.

In Chapter 5, we proposed two new data compression algorithms using an AD-tree and a suffix tree based on antidictionaries. It is proved that both of them work in linear time with respect to string length. The proposed algorithm based on the suffix tree works in linear time with an on-line manner, while the traditional DCA algorithms with an on-line manner require worst case quadratic computational time with respect to the string length. Moreover, we proposed an on-line arithmetic coding using dynamic suffix trees based on antidictionaries. We showed that the time complexity of our algorithm is linear with the string length. It was shown that a tree model based on antidictionaries using dynamic suffix trees provides an efficient statistical model for an adaptive arithmetic coding by simulation results. Experimental results show that the proposed algorithm achieved better compression ratios than the traditional on-line DCA for almost all files on Calgary Corpus, and this approach gives a 3% decrease in compressed file size relative the traditional on-line DCA and a 1% decrease in the size relative to the OHY.

In Chapter 6, we proposed a new on-line lossless data compression for the ECG data using an antidictionary. The proposed algorithm constructs the antidictionary by means of the substring of the ECG signals using the properties such that most of the ECG waves take the similar form of one another. Experimental results showed the proposed algorithm gives 15% decrease in compressed file size relative to the popular compress algorithms such as the LZ algorithm. We showed the proposed algorithm is available for transmitting real-time by computer simulation. Moreover, we studied on the length of substring needed to construct an antidictionary whose size is almost same as that of the entire string of ECG using the results of coupon collector's problems. Experimental results show its validity of estimation.

References

[AHU83] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data structures and algorithms,* Addison-Wesley Publishing, 1983.

[AS92] A. Apostolico and W. Szpankowski, “Self-alignment in words and their applications,” *J. Algorithms*, vol. 13, no. 3, pp. 446–467, 1992.

[Ash90] R. B. Ash, *Information theory,* Dover Publications Inc., 1990.

[BBHECS85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen and J. Seiferas, “The smallest automaton recognizing the subwords of a text,” *Theoretical Computer Science*, vol. 40, no.1, pp. 31–55, 1985.

[BEH89] A. Blumer, A. Ehrenfeucht and D. Haussler, “Average sizes of suffix trees and dawgs,” *Discrete Applied Mathematics*, vol. 24, pp. 37–45, 1989.

[Cal] Calgary text compression corpus. `ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/`

[CEGM04] M. Crochemore, C. Epifanio, R. Grossi and F. Mignosi, “A Trie-Based Approach for Compacting Automata,” *Proc. of the Combinatorial Pattern Matching 15th Annual Symp.*vol. 3109, pp.145–158, July 2004.

[CMR98] M. Crochemore, F. Mignosi and A. Restivo, "Automata and forbidden words," *Information Processing Letters*, vol. 67, no. 3, pp. 111–117, Oct. 1998.

[CMRS00] M. Crochemore, F. Mignosi, A. Restivo and S. Salemi, "Data compression using antidictionaries," *Proc. of the IEEE*, vol. 88, no. 11, pp. 1756–1768, Nov. 2000.

[CN02] M. Crochemore and G. Navarro, "Improved antidictionary based compression," *XII International Conference of the Chilean Computer Science Society (SCCC'02)*, pp. 7–13, Nov. 2002.

[CR02] M. Crochemore and W. Rytter, *Jewels of stringology,* World Scientific, 2002.

[Cro86] M. Crochemore, "Transducers and repetitions," *Theoretical Computer Science*, vol. 45, no.1, pp. 63–86, Sep. 1986.

[CTW95] J. G. Cleary, W. J. Teahan and I. H. Witten, "Unbounded length contexts for PPM," *Proceedings of the IEEE Data Compression Conference (DCC'95)*, pp. 52–61, 1995.

[CW84] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. on Comm.*, vol. 32, no. 4, pp. 396–402, Apr. 1984.

[Dur99] R. Durrett, *Essentials of stochastic processes*, Springer-Verlag, 1999.

[DSR92] L. Devroye, W. Szpankowski and B. Rais, "A note of the height of suffix trees," *SIAM J. Computing*, vol. 21, pp. 48–53, 1992.

[El75] P. Elias, “Universal codeword sets and representations of the integers,” *IEEE Trans. Inform. Theory*, vol.IT-21, no.2, pp.194-203, 1975.

[Fay06] J. Fayolle, “Lossless data compression and analytic combinatorics,” Ph.D. thesis, Pierre & Marie Curie University, France, Mar. 2006.

[Far97] M. Farach-Colton, “Optimal suffix tree construction with large alphabets,” *Proc. of the Foundations of Computer Science (FOCS '97)*, pp.137–143, 1997.

[FG89] E. R. Fiala and D. H. Greece, “Data compression with finite windows,” *Communications of the ACM.* vol. 32, no. 4, pp.490–505, Apr. 1989.

[FMRS06] G. Fici, F. Mignosi, A. Restivo and M. Sciortino, “Word assembly through minimal forbidden words,” *Theoretical Computer Science*, vol. 359, pp.214–230, Aug. 2006.

[GKP89] R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete mathematics*, Addison-Wesley Publishing Company, 1989.

[Gus97] D. Gusfield, *Algorithms on strings, trees, and sequences*, Cambridge University Press, 1997.

[Huf52] D. A. Huffman, “A method for the construction of minimum-redundancy codes,” *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp.1098–1101, Sep. 1952.

[JHSC90] S. M. S. Jalaleddine, C. G. Hutchens, R. D. Strattan and W. A. Coberly, "ECG data compression technique -A unified approach," *IEEE Trans. on Biomed. Eng.*, vol. 37, no. 4, pp.329–343, 1990.

[JLS04] S. Janson, S. Lonardi and W. Szpankowski, "On average sequence complexity," *Proc. of the Combinatorial Pattern Matching*, pp.74–88, 2004.

[JL02] J-L. Gailly, `gzip`, 2002. `http://www.gzip.org/`

[Lar96] N. J. Larsson, "Extended application of suffix trees to data compression," *Proceedings of Data Compression Conference*, pp.190–199, 1996.

[MK92] H. Morita and K. Kobayashi, "Data compression of computer files based on restricted reproducible parsing algorithms," *Trans. on Information Processing Society of Japan*, vol. 33, no. 2, pp.110–121, 1992.

[MK93] H. Morita and K. Kobayashi, "Data compression of ECG based on the edit distance algorithms," *IEICE Trans. on Information and Systems*, vol. E76-D, no. 12, pp.1443–1453, 1993.

[McC76] E. M. McCreight, "A space-economical suffix tree construction algorithm," *J. ACM*, vol. 23, pp. 262–272, 1976.

[MIT] MIT-BIH Arrhythmia Database, `http://www.physionet.org/physiobank/database/mitdb/`

[MO04] H. Morita and T. Ota, "On-line ECG lossless compression using antidictionaries," *Proc. of General Theory of Information Transfer and Combinatorics*, pp. 53–54, Apr. 2004.

[MO05] H. Morita and T. Ota, “A tight upper bound on the size of the antidictionary of a binary string,” *Proc. 2005 Int’l Conf. on the Analysis of Algorithms (AofA’05)*, pp. 393–398, 2005.

[Mof90] A. Moffat, “Implementing the PPM data compression scheme,” *IEEE Trans. on Communications*, vol. 38, no. 11, pp. 1917–1921, 1990.

[MRS02] F. Mignosi, A. Restivo, and M. Sciortino, “Words and forbidden factors,” *Theoretical Computer Science*, vol. 273. pp. 99–117, 2002.

[MS98] H. Morita and H. Sato, “Arithmetic coding,” in *Source coding*, Society of Information Theory and its Applications, Baifukan, 2002.

[MT02] A. Moffat and A. Turpin, *Compression and coding algorithm*, Kluwer Academic Publishers, 2002.

[OHY05] N. Ohkawa, K. Harada and H. Yamamoto, “Data compression by arithmetic coding and source modeling based on the anti-dictionary method,” *IEICE Technical Report*, vol. 105, no. 191, pp. 41–46, July 2005 (in Japanese).

[OM04] T. Ota and H. Morita, “One-pass ECG lossless compression using antidictionaries,” *IEICE Trans. on Fundamentals* (Japanese Edition), vol. J87-A, no. 9, pp. 1187–1195, Sep. 2004 (in Japanese).

[OM05a] T. Ota and H. Morita, “Construction antidictionary of a binary string using suffix tree,” *IEICE Technical Report* , vol. 105, no. 191, pp. 9–14, Jul. 2005 (in Japanese).

[OM05b] T. Ota and H. Morita, "Linear complexity construction of antidictionaries," *Proc. of the 28th Symp. on Information Theory and Its Applications (SITA2005)*, vol. 1, pp. 407–410, Nov. 2005.

[OM06a] T. Ota and H. Morita, "Data compression using source model based on antidictionary tree," *IEICE Technical Report* , vol. 105, no. 661, pp. 135–140, Mar. 2006 (in Japanese).

[OM06b] T. Ota and H. Morita, "On the construction of an antidictionary of a binary string with linear complexity," *Proc. of 2006 IEEE Int'l Symp. on Information Theory (ISIT2006)*, pp.2343–2347, July 2006.

[OM06c] T. Ota and H. Morita, "Data compression using antidictionary for finite-alphabet sources," *IEICE Technical Report*, vol. 106, no. 184, pp. 37–42, July 2006 (in Japanese).

[OM06d] T. Ota and H. Morita, "On-line arithmetic coding based on antidictionaries," *Proc. of the 29th Symp. on Information Theory and Its Applications (SITA2006)*, vol. 2, pp. 513–516, Nov. 2006 (in Japanese).

[OM07a] T. Ota and H. Morita, "On the on-line arithmetic coding based on antidictionaries with linear complexity," *Proc. of 2007 IEEE Int'l Symp. on Information Theory (ISIT2007)*, pp.86–90, June 2007.

[OM07b] T. Ota and H. Morita, "On the construction of an antidictionary with linear complexity using the suffix tree," *IEICE Trans. on Fundamentals*, (in press), Nov. 2007.

[Pas76] R. Pasco, “Source Coding Algorithms for Fast Data Compression,” Ph.D thesis, Stanford University, 1976.

[Ris76] J. J. Rissanen, “Generalized Kraft inequality and arithmetic coding,” *IBM Journal of Research and Development*, vol. 20, pp.198–203, May 1976.

[Ris78] J. J. Rissanen, “Modeling by shortest data description,” *Automatica*, vol. 14, pp.465–471, 1978.

[Ris83] J. J. Rissanen, “A universal data compression system,” *IEEE Transactions on Information Theory*, vol. 29, no. 5, pp.656–664, 1986.

[Sed90] R. Sedgewick, *ALGORITHM in C,* Addison-Wesley Publishing, 1990.

[SF96] R. Sedgewick and P. Flajolet, *Analysis of algorithms*, Addiwon-Wesley, 1996.

[SMN06] D. Sun, H. Morita and M. Nishiara, “On construction of reversible variable-length codes including synchronization markers as codewords,” *IEICE Technical Report*, vol. 105, no. 661, pp. 141–145, Mar. 2006.

[SS82] J. A. Storer and T. G. Szymanski, “Data compression via textual substitution,” *Journal of the ACM.*, vol. 29, no. 4, pp. 928–951, Oct. 1982.

[Ukk95] E. Ukkonen, “On-line construction of suffix-trees,” *Algorithmica*, vol. 14, pp. 249–260, 1995.

[Wei73] P. Weiner, "Linear pattern matching algorithms," *Proc. IEEE 14th Annual Symp. on Switching and Automata Theory*, pp. 1–11, 1973.

[Wel84] T. Welch, "A technique for high-performance data compression," *IEEE Computer*, vol. 17, no. 6, pp. 8–19, June 1984.

[WC87] I. Witten and J. Cleary, "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[WST95] F. M. J. Willems, Y. M. Shtarkov, and T. J. Tjalkens, "The context tree weighting method: Basic properties," *IEEE Trans. on Inf. Theory*, vol. 41, no. 3, pp. 653–664, 1995.

[ZL77] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. on Inf. Theory*, vol.23, no. 3, pp. 337–343, 1977.

[ZL78] J. Ziv and A. Lempel, "Compression of individual sequences via variable length coding," *IEEE Trans. on Inf. Theory*, vol.24, no. 5, pp. 530–536, Sep. 1978.

Acknowledgements

I would like to express my thankfulness to Professor Hiroyoshi Morita, who made it possible for me to become a Ph.D. candidate in computer science. I would like to thank him very much for the straight way of dealing with various issues as well as for all the fruitful discussions and advices that significantly influenced this thesis.

I would like to express my thanks to Professor Kunikatsu Takase, Professor Kenji Tanaka, Professor Mamoru Hoshi and Professor Hiroshi Nagaoka for taking their time and effort to read, evaluate and make useful comments on the thesis.

I would like to express my thanks to thank Dr. Adriaan J. van Wijngaarden. He gave me a lot of help and proofreading the conference manuscripts and the thesis.

I would like to thank Dr. Mototaro Sato, Dr. Tsutomu Otake, Dr. Seiichi Osawa and all the colleagues in Nagano Prefectural Institute of Technology, and Mr. Kei Otsubo, Miss Miki Ishizaki and Mr. Mikio Kobayashi who are graduates from the Institute of Technology, for their support and help.

I would like to thank all the members in the math-sys and the applied network laboratory for their kindness and help, and I would like to thank

Todorka Alexandrova Ph.D. candidate in the laboratory for giving me many valuable comments on presentations in English.

I would like to thank Mr. and Mrs. Katayama for giving me a opportunity for making academic researches.

Lastly, I would like to give my special thanks to my friends, my family, my uncle and my parents for their kindness and help.

Author Biography

Takahiro Ota was born in Nagano, Japan, on November 4, 1970. He received the B.Eng. degree from the University of Electro-Communications, Tokyo, Japan, in 1993. He has been a doctoral candidate with the Graduate School of Information Systems, the University of Electro-Communications from April 2005 to September 2007. In 1993, he joined the Fire and Disaster Prevention Division, Nagano Prefectural Government, Japan. Since 1997, he has been a Lecturer of the Department of Electronic Engineering at the Nagano Prefectural Institute of Technology, Nagano, Japan. His research interests are in information theory, bioinformatics and data compression. He is a student member of the Institute of Electronics, Information and Communication Engineers (IEICE), Society of Information Theory and its Applications (SITA) and Information Processing Society of Japan (IPSJ). He received the Young Researcher Award from SITA in 2006.

List of Publications Related to the Thesis

Journal Papers (refereed)

[1] **T. Ota** and H. Morita, “One-pass ECG lossless compression using antidictionaries,” *IEICE Trans. on Fundamentals* (Japanese Edition), vol. J87-A, no. 9, pp. 1187–1195, Sep. 2004 (in Japanese).
(The contents of Chapter 6)

[2] **T. Ota** and H. Morita, “On the construction of an antidictionary with linear complexity using the suffix tree,” *IEICE Trans. on Fundamentals*, (in press), Nov. 2007.
(The contents of Chapter 4)

International Conference Papers (refereed)

[1] **T. Ota** and H. Morita, “On the construction of an antidictionary of a binary string with linear complexity,” *Proc. of 2006 IEEE International Symposium on Information Theory (ISIT2006)*, pp.2343–2347,

July 2006.

(The contents of Chapter 4)

[2] **T. Ota** and H. Morita, "On the on-line arithmetic coding based on antidictionaries with linear complexity," *Proc. of 2007 IEEE International Symposium on Information Theory (ISIT2007)*, pp.86–90, June 2007.

(The contents of Chapter 5)

Technical Reports

[1] **T. Ota** and H. Morita, "DCA lossless compression and detecting arrhythmia of ECG," *Proc. of the 25th Symposium on Information Theory and Its Applications (SITA2002)*, vol. 2, pp. 551–554, Dec. 2002 (in Japanese).

(The contents of Chapter 6)

[2] **T. Ota** and H. Morita, "Evaluation of length of substrings to construct antidictionary used in lossless ECG compression," *Proc. of the 26th Symposium on Information Theory and Its Applications (SITA2003)*, vol. 1, pp. 65–68, Dec. 2003 (in Japanese).

(The contents of Chapter 6)

[3] **T. Ota** and H. Morita, "Construction antidictionary of a binary string using suffix tree," *IEICE Technical Report* , vol. 105, no. 191, pp. 9–14, Jul. 2005 (in Japanese).

(The contents of Chapter 4)

[4] **T. Ota** and H. Morita, "Linear complexity construction of antidictionaries," *Proc. of the 28th Symposium on Information Theory and Its Applications (SITA2005)*, vol. 1, pp. 407–410, Nov. 2005.
(The contents of Chapter 4)

[5] **T. Ota** and H. Morita, "Data compression using source model based on antidictionary tree," *IEICE Technical Report* , vol. 105, no. 661, pp. 135–140, Mar. 2006 (in Japanese).
(The contents of Chapter 5)

[6] **T. Ota** and H. Morita, "Data compression using antidictionary for finite-alphabet sources," *IEICE Technical Report*, vol. 106, no. 184, pp. 37–42, July 2006 (in Japanese).
(The contents of Chapter 2, 3, 4 and 5)

[7] **T. Ota** and H. Morita, "On-line arithmetic coding based on antidictionaries," *Proc. of the 29th Symposium on Information Theory and Its Applications (SITA2006)*, vol. 2, pp. 513–516, Nov. 2006 (in Japanese).
(The contents of Chapter 5)

Related Works

[1] H. Morita and **T. Ota**, "On-line ECG lossless compression using antidictionaries," *Proc. of General Theory of Information Transfer and Combinatorics*, pp. 53–54, Apr. 2004.
(The contents of Chapter 6)

[2] H. Morita and **T. Ota**, “A tight upper bound on the size of the antidictionary of a binary string,” *Proc. 2005 International Conference on the Analysis of Algorithms (AofA’05)*, pp. 393–398, June 2005.
(The contents of Chapters 2 and 4)

Award

[1] SITA Young Researcher Award, “Linear complexity construction of antidictionaries,” *Society of Information and its Applications(SITA)*, 2006.
(The contents of Chapter 4)

List of Figures

List of Tables