Studies on Efficient Index Construction for Multiple and Repetitive Texts

(複数テキストと繰り返しテキストに対する効率の良い索引構築の研究)

Takuya Takagi

January 2018

Division of Computer Science and Information Technology
Graduate School of Information Science and Technology
Hokkaido University

Abstract

Text indexing problem is one of the fundamental problems in computer science and the aim is to construct an efficient data structure that answers queries such as text pattern matching. For the last decades, there has been an increasing amount of *multiple texts* such as data generated from multiple sensors and *repetitive texts* such as genome sequence collections. For example, the *GeoLife Project* collects trajectories from GPS loggers that have a variety of sampling rates. These trajectories were recorded every 1 to 5 seconds or every 5 to 10 meters per point. For another example, the *1000 Genomes Project* collects the human genomes from various groups. Since each genome information is similar to each other, the same substructures appear repeatedly in this genome database. These projects are aiming at data analysis, information retrieval, and data mining for text information. For pattern matching, which is the most fundamental query for texts, we can answer queries by using basic text pattern matching algorithms such as Knuth-Morris-Pratt (KMP) algorithm and Boyer-Moore (BM) algorithm. Since these algorithms scan the texts for each query, it requires at least linear time for database size in one query. In order to quickly process these data, preprocessing and *indexing* are important. For example, the suffix tree, one of the basic text indexes, can support pattern matching in linear time for pattern length. Therefore, building an efficient index structure is the key to processing these large amounts of text information. In this thesis, we show efficient index construction algorithms for text data.

For multiple texts and repetitive texts, there are several problems with indexing.

Since data grow constantly for multiple sensor data such as GPS trajectories, it is necessary for the index to support online construction for multiple texts. For repetitive texts that is similar text collection such as genome sequences, we should be able to build an index with a more compressed size. In order to solve these problems, we propose several new index structures and construction algorithms. In particular, this thesis deals with speeding up construction and operations of indexes, online construction of indexes for multiple texts, and construction of compressed indexes for texts including long repetitions.

In Chapter 3, we propose a faster version of labeled trees (compact tries) called *packed compact tries*, by using a bit-parallel method. By doing this, we show faster construction of text indexes such as suffix trees and faster various operations like prefix search, insertion, and deletion. Since the compact trie is a widely used data structure, we can speed up some algorithms by using packed compact tries. In particular, we show that LZ-double factorization which is one kind of text compression algorithm is speeded up.

In Chapter 4, we first defined a fully-online construction problem, which is a setting that allows a new input symbol can be added an arbitrary string of the set of input strings. To solve this problem, we first showed a fully-online construction algorithm of a DAG index called the *directed acyclic word graph* (DAWG). We also proposed a fully-online construction algorithm for the suffix tree using similarity between DAWGs and suffix trees.

In Chapter 5, we proposed a self-indexing method by combining an index called the *compact directed acyclic word graph (CDAWG)* with grammar compression, which is one of the compression methods. When the input text is compressible, the index can be held with a size smaller than the original text.

In Chapter 6, we give conclusions and future work. Overall, we studied efficient algorithms for text index construction in this thesis.

Contents

Chapter 1

Introduction

1.1 Background

For the last decades, there has been an increasing amount of unstructured data such as genetic data, logging data, and Web and SNS texts, which have been coined as *big data*. Most of these unstructured data are available in the form of text information. Therefore, there are demands for algorithms and data structure that can efficiently handle these big unstructured data. *Multiple growing texts* and *repetitive texts* are one of the features of text data such as logging data and genetic data.

Multiple growing texts are a text set that can be appended a new symbol at the end of a text in the set. There are many text data with this feature in the real world. For example, due to the rapid development of network and sensor technologies, various and enormous *stream data* are generated from multiple source such as GPS trajectory data [72], sensor and Twitter streams. These are represented as *multiple texts* or *multiple sequences* that are constantly growing.

Another feature of textual big data is called *repetitiveness* [56]. It means a kind of text sets consisting of similar texts. For example, genome sequences [21] and versioned document collections such as software repositories are one of the *highly repetitive texts*.

These data contain many long repetitions in the text.

1.2 Research goals

In order to use big data, it is necessary to perform various queries such as data mining and information retrieval. However, because of the massive amount of data, even simple queries such as text pattern matching take too much time. One of the solutions is to preprocess those data and create an index that supports the query in order to answer quickly. Among indexes for texts, those indexes that have all substring information of each text supports the most diverse queries.

In this thesis, we study efficient index construction for multiple texts and repetitive texts. There are the following demands for construction of indexes with these text data. First, in order to process a large amount of data at high speed, we want an index that supports fast queries. Second, in order to construct an index for multiple growing texts, we need an index that enables online construction for multiple texts. Finally, to store a large amount of data, it must be small in its size.

1.3 Summary of the results

In this thesis, there are three main results to text indexing as follows. In Chapter 2, we introduce notations and definitions of some data structures.

In Chapter 3, we study acceleration of compact tries using the packed string technique. The *dynamic compact trie* [42, 65] is a fundamental data structure for storing a set of variable-length strings. It can store a set of k strings over an alphabet Σ with total size n in $O(n \log n)$ bits of space. we propose *packed compact tries* that support faster prefix search queries and update operations of compact tries on the standard word RAM model. It still keeps $n \log \sigma + O(k \log n)$ bits of space.

In Chapter 4, we study *fully-online* construction of DAWG and suffix trees for multiple texts. Let $\mathcal{T} = \{T_1, \ldots, T_K\}$ be a collection of texts. By fully-online, we mean that a new character can be appended to any text in $\mathcal{T}$ *at any time*. This is a natural generalization of *semi-online* construction of indexing data structures for multiple texts in which, after a new character is appended to the k-th text T_k, then its previous texts $T_1, \ldots, T_{k-1}$ will remain static. We propose fully-online algorithms which construct the *directed acyclic word graph* (*DAWG*) [14], and the *generalized suffix tree* (*GST*) [42] for $\mathcal{T}$ in $O(n \log \sigma)$ time and $O(n)$ words of space, where n and σ denote the total length of texts in $\mathcal{T}$ and the alphabet size, respectively.

In Chapter 5, we study a compressed index combining CDAWGs and grammar compression. Recent studies have shown that the compact directed acyclic word graphs (CDAWG) [15] topology achieves the compressed size for repeated strings. However, there is no known method for supporting high-speed search with the compressed size without having the original input string. Linear-size CDAWG proposed in this thesis achieves the compressed size while supporting search time similar to original CDAWG.

In Chapter 6, we give the summary of this thesis, and then discuss possible future researches.

1.4 Contributions of this thesis

We studied three fundamental problems which are necessary when we construct the index that can efficiently handle a massive amount of text data. A versatile text index has three features: high speed queries, fully-online construction, and, small space complexity. Each result of this thesis shows an index which achieves one of the three features. First, as a basis of efficient text indexes allowing high speed query processing, we proposed an improved data structure supporting high speed construction and queries by using bit-parallel methods. Secondly, for multiple growing texts like stream data

from multiple sensors, we proposed construction algorithm of an index in fully-online manner. Thirdly, for texts that contain many repetitive structures, we proposed an index that can capture the repeating structure and store it in compressed size. Overall, we studied efficient algorithms for text index construction which are a basis to achieve an index with the three features.

Chapter 2

Preliminaries

In this chapter, we introduce basic definitions and notations in strings, suffix tries, suffix trees, directed acyclic word graphs, and compact directed acyclic word graphs according to [24–26, 42].

2.1 Notations on Strings

Let Σ be an ordered alphabet. Any element of Σ^* is called a *string*. For any string T, let $|T|$ denote its length. Let ε be the empty string, namely, $|\varepsilon| = 0$. If $T = XYZ$, then X, Y, and Z are called a *prefix*, a *substring*, and a *suffix* of T, respectively. For any $1 \leq i \leq j \leq |T|$, let $T[i..j]$ denote the substring of T that begins at position i and ends at position j in T. For any $1 \leq i \leq |T|$, let $T[i]$ denote the ith character of T. For any string T, let $\mathit{Suffix}(T)$ denote the set of suffixes of T, and for any set $\mathcal{T}$ of strings, let $\mathit{Suffix}(\mathcal{T})$ denote the set of suffixes of all strings in $\mathcal{T}$. Namely, $\mathit{Suffix}(\mathcal{T}) = \bigcup_{T \in \mathcal{T}} \mathit{Suffix}(T)$. For any string T, let $\overline{T}$ denote the reversed string of T, i.e., $\overline{T} = T[|T|] \cdots T[1]$.

Let $\mathcal{T} = \{T_1, \ldots, T_K\}$ be a collection of K texts. For any $1 \leq k \leq K$, let $\mathit{lrs}_{\mathcal{T}}(T_k)$ be the longest repeating suffix of T_k that occurs at least twice in $\mathcal{T}$. For any strings

X, Y, $LCP(X, Y)$ denotes the longest common prefix of X and Y.

Throughout this thesis, the base of the logarithms will be 2, unless otherwise stated. For any integers $i \leq j$, $[i, j]$ denotes the interval $\{i, i+1, \ldots, j\}$. Our model of computation is the standard word RAM of word size $w = \log n$ bits. For simplicity, we assume that w is a multiple of $\log \sigma$, so $\alpha = \log_\sigma n$ letters are packed in a single word. Since we can read w bits in constant time, we can read and process α consecutive letters in constant time.

2.2 Notations on graphical indexes

All index structures dealt with in this thesis, such as suffix tries, suffix trees, CDAWGs, linear-size suffix tries (LSTries), and linear-size CDAWGs (L-CDAWGs), are *graphical indexes* in the sense that an index is a pointer-based structure built on an underlying DAG $G_L = (V(L), E(L))$ with a root $r \in V(L)$ and mapping $lab : E(L) \to \Sigma^+$ that assign a label $\mathrm{id}lab(e)$ to each edge $e \in E(L)$. For an edge $e = (u, v) \in E(L)$, we denote its *end points* by $e.hi := u$ and $e.lo := v$, respectively. The *label string* of e is $\mathrm{id}lab(e) \in \Sigma^+$. The *string length* of e is $\mathrm{id}slen(e) := |\mathrm{id}lab(e)| \geq 1$. An edge is called *atomic* if $\mathrm{id}slen(e) = 1$, and thus, $\mathrm{id}lab(e) \in \Sigma$. For a path $p = (e_1, \ldots, e_k)$ of length $k \geq 1$, we extend its *end points*, *label string*, and *string length* by $p.hi := e_1.hi$, $p.lo := e_k.lo$, $\mathrm{id}lab(p) := \mathrm{id}lab(e_1) \ldots \mathrm{id}lab(e_k) \in \Sigma^+$, and $\mathrm{id}slen(p) := \mathrm{id}slen(e_1) + \cdots + \mathrm{id}slen(e_k) \geq 1$, respectively.

2.3 Suffix tries

The *suffix trie* for a text collection $\mathcal{T} = \{T_1, \ldots, T_K\}$, denoted $STrie(\mathcal{T})$, is a trie which represents $Suffix(\mathcal{T})$. The size of $STrie(\mathcal{T})$ is $O(n^2)$, where n is the total length of texts in $\mathcal{T}$. We identify each node v of $STrie(\mathcal{T})$ with the string that v represents.

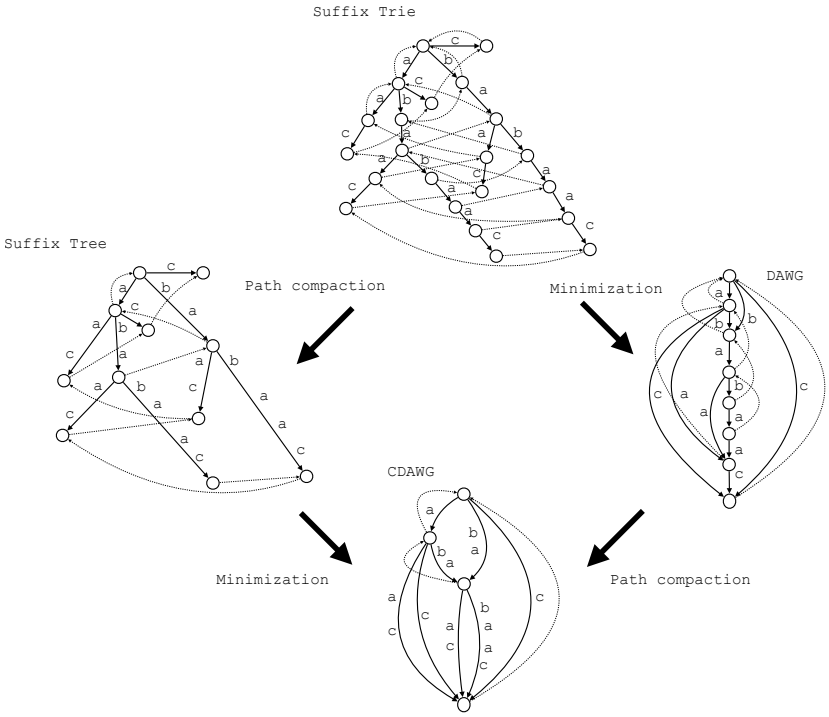

Figure 2.1: Illustration of $STrie(T)$, $STree(T)$, $DAWG(\mathcal{T})$, and $CDAWG(T)$ with $T = \texttt{ababaac}$. The solid arrows and broken arrows represent the edges and the suffix links of each data structure, respectively.

A substring x of a text in $\mathcal{T}$ is said to be *branching* in $\mathcal{T}$, if there exist two distinct characters $a, b \in \Sigma$ such that both xa and xb are substrings of some texts in $\mathcal{T}$. Clearly, node x of $STrie(\mathcal{T})$ is branching iff x is branching in $\mathcal{T}$.

For each node av of $STrie(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, let $slink(av) = v$. This auxiliary edge $slink(av) = v$ from av to v is called a *suffix link*. We define the *reversed suffix link* $\mathcal{W}_a(v) = av$ iff $slink(av) = v$. For any node v and $a \in \Sigma$, if av is not a substring of the texts in $\mathcal{T}$, then $\mathcal{W}_a(v)$ is undefined. By definition, the reversed suffix links on $STrie(\mathcal{T})$ form a rooted tree which coincides with $STrie(\overline{\mathcal{T}})$, the suffix trie for the collection $\overline{\mathcal{T}} = \{\overline{T_1}, \ldots, \overline{T_K}\}$ of the reversed texts.

2.4 Suffix trees

The *suffix tree* [68] for a text collection $\mathcal{T}$, denoted $STree(\mathcal{T})$, is a "compacted trie" which represents $Suffix(\mathcal{T})$. $STree(\mathcal{T})$ is obtained by compacting every path of $STrie(\mathcal{T})$ which consists of non-branching internal nodes (see Fig. 2.1). Since every internal node of $STree(\mathcal{T})$ is branching, and since there are at most n leaves in $STree(\mathcal{T})$, the numbers of edges and nodes are $O(n)$. The edge labels of $STree(\mathcal{T})$ are non-empty substrings of some text in $\mathcal{T}$. By representing each edge label x with a triple $\langle k, i, j\rangle$ of integers s.t. $x = T_k[i..j]$, $STree(\mathcal{T})$ can be stored with $O(n)$ space. We say that any branching (resp. non-branching) substring of $\mathcal{T}$ is an *explicit node* (resp. *implicit node*) of $STree(\mathcal{T})$. An implicit node x is represented by a triple (v, a, ℓ), called a *reference* to x, such that v is an explicit ancestor of x, a is the first character of the path from v to x, and ℓ is the length of the path from v to x. A reference (v, a, ℓ) to node x is called *canonical* if v is the lowest explicit ancestor of x.

For each explicit node av of $STree(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, let $slink(av) = v$. For each explicit node v and $a \in \Sigma$, we also define the reversed suffix link $\mathcal{W}_a(v) = avx$ where $x \in \Sigma^*$ is the shortest string such that avx is an explicit node of $STree(\mathcal{T})$. $\mathcal{W}_a(v)$ is undefined if av is not a substring of texts in $\mathcal{T}$. These reversed suffix links are also called as *Weiner links* (or *W-link* in short) in the literature [16]. A W-link $\mathcal{W}_a(v) = avx$ is said to be *hard* if $x = \varepsilon$, and *soft* if $x \in \Sigma^+$. Let w be a Boolean function such that for any explicit node v and $a \in \Sigma$, $\mathsf{w}_a(v) = 1$ iff (soft or hard) W-link $\mathcal{W}_a(v)$ exists. Notice that if $\mathsf{w}_a(v) = 1$ for a node v and $a \in \Sigma$, then $\mathsf{w}_a(u) = 1$ for every ancestor of v.

2.5 Directed acyclic word graphs (DAWGs)

The *directed acyclic word graph* (*DAWG* in short) [14,15] of a text collection $\mathcal{T}$, denoted $DAWG(\mathcal{T})$, is a smallest DAG which represents $Suffix(\mathcal{T})$. $DAWG(\mathcal{T})$ is obtained by

merging identical subtrees of $STrie(\mathcal{T})$ connected by the suffix links (see Fig. 2.1). Hence, the label of every edge of $DAWG(\mathcal{T})$ is a single character. The numbers of nodes and edges of $DAWG(\mathcal{T})$ are $O(n)$ [15], and hence $DAWG(\mathcal{T})$ can be stored with $O(n)$ space. $DAWG(\mathcal{T})$ can be defined formally as follows: For any string x, let $Epos_{\mathcal{T}}(x)$ be the set of ending positions of x in the texts in $\mathcal{T}$, i.e.,

$$Epos_{\mathcal{T}}(x) = \{(k, j) \mid x = T_k[j - |x| + 1..j], 1 \leq j \leq |T_k|, 1 \leq k \leq K\}.$$

Consider an equivalence relation $\equiv_{\mathcal{T}}$ on substrings x, y of texts in $\mathcal{T}$ such that $x \equiv_{\mathcal{T}} y$ iff $Epos_{\mathcal{T}}(x) = Epos_{\mathcal{T}}(y)$. For any substring x of texts of $\mathcal{T}$, let $[x]_{\mathcal{T}}$ denote the equivalence class w.r.t. $\equiv_{\mathcal{T}}$. There is a one-to-one correspondence between each node v of $DAWG(\mathcal{T})$ and each equivalence class $[x]_{\mathcal{T}}$, and hence we will identify each node v of $DAWG(\mathcal{T})$ with its corresponding equivalence class $[x]_{\mathcal{T}}$. Let $long([x]_{\mathcal{T}})$ denote the longest member of $[x]_{\mathcal{T}}$. By the definition of equivalence classes, $long([x]_{\mathcal{T}})$ is unique for each $[x]_{\mathcal{T}}$ and every member of $[x]_{\mathcal{T}}$ is a suffix of $long([x]_{\mathcal{T}})$. If x, xa are substrings of some text in $\mathcal{T}$ with $x \in \Sigma^*$ and $a \in \Sigma$, then there exists an edge labeled with character $a \in \Sigma$ from node $[x]_{\mathcal{T}}$ to node $[xa]_{\mathcal{T}}$. This edge is called *primary* if $|long([x]_{\mathcal{T}})| + 1 = |long([xa]_{\mathcal{T}})|$, and is called *secondary* otherwise. For each node $[x]_{\mathcal{T}}$ of $DAWG(\mathcal{T})$ with $|x| \geq 1$, let $slink([x]_{\mathcal{T}}) = y$, where y is the longest suffix of $long([x]_{\mathcal{T}})$ which does not belong to $[x]_{\mathcal{T}}$. In the example of Fig. 2.1, $[\texttt{aaab}]_{\mathcal{T}} = \{\texttt{aaab}, \texttt{aab}\}$. The edge labeled with $\texttt{b}$ from node $[\texttt{aaa}]_{\mathcal{T}}$ to node $[\texttt{aaab}]_{\mathcal{T}}$ is primary, while the edge labeled with $\texttt{b}$ from $[\texttt{aa}]_{\mathcal{T}}$ to node $[\texttt{aaab}]_{\mathcal{T}}$ is secondary. $slink([\texttt{aaab}]_{\mathcal{T}}) = [\texttt{ab}]_{\mathcal{T}}$.

2.6 Duality of suffix trees and DAWGs

There exists a nice duality between suffix trees and DAWGs. To observe this, it is convenient to consider the collection $\overline{\mathcal{T}}$ of the reversed texts each of which begins with a special marker $\$_i$, i.e., $\overline{\mathcal{T}} = \{\$_1\overline{T_1}, \ldots, \$_K\overline{T_K}\}$. For ease of notation, let $S_k = \overline{\mathcal{T}_k}$ for

$1 \leq k \leq K$ and $\mathcal{S} = \{\$_1 S_1, \ldots, \$_K S_K\} = \overline{\mathcal{T}}$. Then, it is known (c.f. [14, 15, 25]) that the reversed suffix links of $DAWG(\mathcal{S})$ coincide with the suffix tree $STree(\mathcal{T})$ for the original text collection $\mathcal{T}$. This fact can also be observed from the other direction. Namely, the hard (resp. soft) W-links of $STree(\mathcal{T})$ coincide with the primary (resp. secondary) edges of $DAWG(\mathcal{S})$.

Intuitively, this duality holds because

(1) The reversed suffix links of $STrie(\mathcal{S})$ form $STrie(\mathcal{T})$ (and vice versa), and

(2) When we construct $DAWG(\mathcal{S})$ from $STrie(\mathcal{S})$, we merge isomorphic subtrees that are connected by suffix links. During this merging process, the reversed suffix links get compacted and the resulting compacted links form the edges of $STree(\mathcal{T})$.

Using this duality, we can immediately show that the total number of hard and soft W-links is linear in the total text length n, since the number of edges of the DAWG is linear in n. This also means that we can easily maintain the Boolean indicator w with $O(n)$ space, so that $\mathsf{w}_a(v)$ for a given node v and $a \in \Sigma$ can be answered in $O(\log \sigma)$ time (e.g., at each node v we can maintain a BST storing only the characters c s.t. $\mathsf{w}_c(v) = 1$.)

2.7 Compact directed acyclic word graphs (CDAWGs)

The *compact directed acyclic word graph* [15, 26] for a text T, denoted $CDAWG(T)$, is the minimal compact automaton which represents $Suffix(T)$. $CDAWG(T)$ can be obtained from $STree(T\$)$ by merging isomorphic subtrees and deleting associated end-marker $\$ \notin \Sigma$. Since $CDAWG(T)$ is an edge-labeled DAG, we represent a directed edge from node u to v with label string $x \in \Sigma^+$ by a triple $f = (u, x, v)$. For any node u, the label strings of out-going edges from u start with mutually distinct characters.

Formally, $CDAWG(T)$ is defined as follows. For any strings x, y, we denote $x \equiv_L y$ (resp. $x \equiv_R y$) iff the beginning positions (resp. ending positions) of x and y in T are equal. Let $[x]_L$ (resp. $[x]_R$) denote the equivalence class of strings w.r.t. $\equiv_L$ (resp. $\equiv_R$). All strings that are *not* substrings of T form a single equivalence class, and in the sequel we will consider only the substrings of T. Let $\overrightarrow{x}$ (resp. $\overleftarrow{x}$) denote the longest member of the equivalence class $[x]_L$ (resp. $[x]_R$). Notice that each member of $[x]_L$ (resp. $[x]_R$) is a prefix of $\overrightarrow{x}$ (resp. a suffix of $\overleftarrow{x}$). Let $\overleftrightarrow{x} = \overleftarrow{(\overrightarrow{x})} = \overrightarrow{(\overleftarrow{x})}$. We denote $x \equiv y$ iff $\overleftrightarrow{x} = \overleftrightarrow{y}$, and let $[x]$ denote the equivalence class w.r.t. $\equiv$. The longest member of $[x]$ is $\overleftrightarrow{x}$ and we will also denote it by $value([x])$. We define $CDAWG(T)$ as an edge-labeled DAG (V, E) such that $V = \{[\overrightarrow{x}]_R \mid x \text{ is a substring of } T\}$ and $E = \{([\overrightarrow{x}]_R, \alpha, [\overrightarrow{x}\alpha]_R) \mid \alpha \in \Sigma^+, \overrightarrow{x} \not\equiv \overrightarrow{x}\alpha\}$. The $\overrightarrow{\cdot}$ operator corresponds to compacting non-branching edges (like conversion from $STrie(T)$ to $STree(T)$) and the $[\cdot]_R$ operator corresponds to merging isomorphic subtrees of $STree(T)$. For simplicity, we abuse notation so that when we refer to a node of $CDAWG(T)$ as $[x]$, this implies $x = \overrightarrow{x}$ and $[x] = [\overrightarrow{x}]_R$.

Let $[x]$ be any node of $CDAWG(T)$ and consider the suffixes of $value([x])$ which correspond to the suffix tree nodes that are merged when transformed into the CDAWG. We define the *suffix link* of node $[x]$ by $slink([x]) = [y]$, iff y is the longest suffix of $value([x])$ that does not belong to $[x]$.

It is shown that all nodes of $CDAWG(T)$ except the sink correspond to the maximal repeats of T. Actually, $value([x])$ is a maximal repeat in T [58]. Following this fact, one can easily see that the numbers of edges of $CDAWG(T)$ and $CDAWG(\overline{T})$ coincide with the numbers e_T^r and e_T^ℓ of right- and left- extensions of maximal repeats of T, respectively [9, 58].

By representing each edge label α with pairs (i, j) of integers such that $T[i..j] = \alpha$, $CDAWG(T)$ can be stored in $O(e_T^r \log n + n \log \sigma)$ bits of space.

Chapter 3

Packed Compact Tries

In this chapter, we present a new data structure called the *packed compact trie* (*packed c-trie*) which stores a set S of k strings of total length n in $n \log \sigma + O(k \log n)$ bits of space and supports fast pattern matching queries and updates, where σ is the alphabet size. Assume that $\alpha = \log_\sigma n$ letters are packed in a single machine word on the standard word RAM model, and let $f(k, n)$ denote the query and update times of the dynamic predecessor/successor data structure of our choice which stores k integers from universe $[1, n]$ in $O(k \log n)$ bits of space. Then, given a string of length m, our packed c-tries support pattern matching queries and insert/delete operations in $O(\frac{m}{\alpha} f(k, n))$ worst-case time and in $O(\frac{m}{\alpha} + f(k, n))$ expected time. Our experiments show that our packed c-tries are faster than the standard compact tries (a.k.a. Patricia trees) on real data sets. As an application of our packed c-trie, we show that the sparse suffix tree for a string of length n over prefix codes with k sampled positions, such as evenly-spaced and word delimited sparse suffix trees, for a set of k word suffixes can be constructed online in $O((\frac{n}{\alpha} + k) f(k, n))$ worst-case time and $O(\frac{n}{\alpha} + k f(k, n))$ expected time with $n \log \sigma + O(k \log n)$ bits of space. When $k = O(\frac{n}{\alpha})$, by using the state-of-the-art dynamic predecessor/successor data structures, we obtain sub-linear time construction algorithms using only $O(\frac{n}{\alpha})$ bits of space in both cases. We also

discuss an application of our packed c-tries to online LZD factorization.

3.1 Background

The trie for a set S of strings of total length n is a classical data structure which occupies $O(n \log n + n \log \sigma)$ bits of space and allows for prefix search and insertion/deletion for a given string of length m in $O(m \log \sigma)$ time, where σ is the alphabet size. The *compact trie* for S is a path-compressed trie where the edges in every non-branching path are merged into a single edge [53]. By representing each edge label by a pair of positions in a string in S, the compact trie can be stored in $n \log \sigma + O(k \log n)$ bits of space, where k is the number of strings in S, retaining the same time efficiency for prefix search and insertion/deletion for a given string. Thus, compact tries have widely been used in numerous applications such as dynamic dictionary matching [44], suffix trees [68], sparse suffix trees [47], external string indexes [30], and grammar-based text compression [39].

In this chapter, we show how to accelerate prefix search queries and update operations of compact tries on the standard word RAM model with machine word size $w = \log n$, still keeping $n \log \sigma + O(k \log n)$-bit space usage. A basic idea is to use the *packed string matching* approach [12], where $\alpha = \log_\sigma n$ consecutive letters are packed in a single word and can be manipulated in $O(1)$ time. In this setting, we can read a given pattern P of length m in $O(\frac{m}{\alpha})$ time, but, during the traversal of P over a compact trie, there can be at most m branching nodes. Thus, a naïve implementation of a compact trie takes $O(\frac{m}{\log_\sigma n} + m \log \sigma) = O(m \log \sigma)$ time even in the packed matching setting.

To overcome the above difficulty, we propose how to quickly process long non-branching paths using bit manipulations, and how to quickly process dense branching subtrees using fast predecessor/successor queries and dictionary look-ups. As a result,

we obtain a new compact trie called the *packed compact trie* (*packed c-trie*) for a dynamic set S of strings with the following efficiency:

Theorem 1 (main result) *Let $f(k,n)$ be the query/update times of an arbitrary dynamic predecessor/successor data structure using $O(k \log n)$ bits of space for a dynamic set of k integers from the universe $[1,n]$. Our packed c-trie stores a set S of k strings of total length n in $n \log \sigma + O(k \log n)$ bits of space and supports prefix search and insertion/deletion for a given string of length m in $O(\frac{m}{\alpha} f(k,n))$ worst-case time or in $O(\frac{m}{\alpha} + f(k,n))$ expected time.*

Using Beame and Fich's data structure [6] or Willard's y-fast trie [70] as the dynamic predecessor/successor data structure, we obtain the following corollary:

Corollary 2 *There exists a packed c-trie for a dynamic set S of strings which uses $n \log \sigma + O(k \log n)$ bits of space, and supports prefix search and insert/delete operations for a given string of length m in $O(\frac{m}{\alpha} \cdot \frac{\log \log k \log \log n}{\log \log \log n})$ worst-case time or in $O(\frac{m}{\alpha} + \log \log n)$ expected time.*

Unlike most other (compact) tries, our packed c-trie does *not* maintain a dictionary or a search structure for the children of each node. Instead, we partition our c-trie into $\lceil h/\alpha \rceil$ levels, where h is the length of the longest string in S. Then each subtree of height α, called a *micro* c-trie, maintains a predecessor/successor dictionary that processes prefix search inside the micro c-trie. A reduction from prefix search to predecessor/successor queries was already considered in an earlier work by Cole et al. [19], however, their data structure is static. On the other hand, our micro c-tries are dynamic. A similar technique to our packed c-trie was used in the linked dynamic uncompacted trie by Jansson et al. [46].

Our experiments show that our packed c-tries are faster than Patricia trees for both construction and prefix search in almost all data sets we tested.

We show that our packed c-tries can be applied to efficient online construction of *evenly sparse suffix trees* [47], *word suffix trees* [45] and its extension [64]. Also, packed c-tries can be used for online computation of the *LZ-Double factorization* [39] (*LZDF*), a state-of-the-art online grammar-based text compressor.

We also show two applications to our packed c-tries. The first application is online construction of *evenly sparse suffix trees* [47], *word suffix trees* [45] and its extension [64]. The existing algorithms for these sparse suffix trees take $O(n \log \sigma)$ worst-case time using $n \log \sigma + O(k \log n)$ bits of where k is the number of suffixes stored in the output sparse suffix tree. Using our packed c-tries, we achieve $O((\frac{n}{\alpha} + k)\frac{\log \log k \log \log n}{\log \log \log n})$ worst-case construction time and $O(\frac{n}{\alpha} + k \log \log n)$ expected construction time. The former is sublinear in n when $k = O(\frac{n}{\alpha})$ and $\sigma = \text{polylog}(n)$, the latter is sublinear in n when $k = o(\frac{n}{\log \log n})$ and $\sigma = \text{polylog}(n)$. To achieve these results, we show that in our packed c-trie, prefix searches and insertion operations can be started not only from the root but from *any* node. This capability is necessary for online sparse suffix tree construction, since during the suffix link traversal we have to insert new leaves from non-root internal nodes.

The second application is online computation of the *LZ-Double factorization* [39] (*LZDF*), a state-of-the-art online grammar-based text compressor. Goto et al. [39] presented a Patricia-tree based algorithm which computes the LZDF of a given string T of length n in $O(k(M + \min\{k, M\} \log \sigma))$ worst-case time using $O(n \log \sigma)$ bits of space, where $k \leq n$ is the number of factors and $M \leq n$ is the length of the longest factor. Using our packed c-tries, we achieve a good expected performance with $O(k(\frac{M}{\alpha} + f(k, n)))$ time for LZDF.

3.1.1 Related work

Belazzougui et al. [7] proposed a *randomized* compact trie called the *signed dynamic z-fast trie*, which stores a dynamic set S of k strings in $n \log \sigma + O(k \log n)$ bits of

space. Given a string of length m, the signed dynamic z-fast trie supports prefix search in $O(\frac{m}{\alpha} + \log m)$ worst-case time *only with high probability*, and supports insert/delete operations in $O(\frac{m}{\alpha} + \log m)$ expected time *only with high probability*.[1] On the other hand, our packed c-trie always return the correct answer for prefix search, and always insert/delete a given string correctly, in the bounds stated in Theorem 1 and Corollary 2.

Andersson and Thorup [3] proposed the *exponential search tree* which uses $n \log \sigma + O(k \log n)$ bits of space, and supports prefix search and insert/delete operations in $O(m + \sqrt{\frac{\log k}{\log \log k}})$ worst-case time. Each node v of the exponential search tree stores a constant-time look-up dictionary for some children of v and a dynamic predecessor/successor data structure for the other children of v. This implies that given a string of length m, at most m nodes in the search path for the string must be processed one by one, and hence packing $\alpha = \log_\sigma n$ letters in a single word does not seem to speed-up the exponential search tree.

Fischer and Gawrychowski's *wexponential search tree* [33] proposed uses $n \log \sigma + O(k \log n)$ bits of space, and supports prefix search and insert/delete operations in $O(m + \frac{(\log \log \sigma)^2}{\log \log \log \sigma})$ worst-case time. When $\sigma = \text{polylog}(n)$, our packed c-trie achieves $O(m \frac{\log \sigma \log \log k \log \log n}{\log n \log \log \log n}) = O(m \frac{(\log \log n)^2}{\log n \log \log \log n}) = O(o(1)m)$ worst-case time, while the wexponential search tree requires $O(m + \frac{(\log \log \log n)^2}{\log \log \log \log n})$ time[2].

[1] The $O(\log m)$ expected bound for insertion/deletion stated in [7] assumes that the prefix search for the string has already been performed.

[2] For sufficiently long patterns of length $m = \Theta(n)$, our packed c-trie achieves worst-case *sublinear* $o(n)$ time while the wexponential search tree requires $O(n)$ time.

3.2 Preliminaries

3.2.1 Compact tries

Let $S = \{X_1, \ldots, X_k\}$ be a set of k non-empty strings of total length n. We consider dynamic data structures for S allowing for fast prefix searches of given patterns over strings in S, and fast insertion/deletion of strings to/from S.

Suppose S is prefix-free. The *trie* of S is a tree s.t. each edge is labeled by a single letter, the labels of the out edges of each node are distinct, and for each $X_i \in S$ there is a unique leaf ℓ_i s.t. the path from the root to ℓ_i spells out X_i.

The *compact trie* $\mathcal{T}_S$ of S is a path-compressed trie obtained by contracting non-branching paths into single edges. Namely, in $\mathcal{T}_S$, each edge is labeled by a non-empty substring of T, each internal node has at least two children, the out-going edges from each node begin with distinct letters, and each edge label x is encoded by a triple $\langle i, a, b\rangle$ such that $x = X_i[a..b]$ for some $1 \leq i \leq k$ and $1 \leq a \leq b \leq |X_i|$. The *length* of an edge e, denoted $|e|$, is the length of its label string. Let $root(\mathcal{T}_S)$ denote the root of the compact trie $\mathcal{T}_S$. For any node v, let $parent(v)$ denotes its parent. For convenience, let $\perp$ be an auxiliary node s.t. $parent(root(\mathcal{T}_S)) = \perp$. We assume the edge from $\perp$ to $root(\mathcal{T}_S)$ is labeled by an arbitrary letter. For any node v, let $str(v)$ denotes the string obtained by concatenating the edge labels from the root to v. Each node v stores $|str(v)|$.

Let s be a prefix of any string in S. Let v be the shallowest node of $\mathcal{T}_S$ such that s is a suffix of $str(v)$ (notice s can be equal to $str(v)$), and let $u = parent(v)$. The *locus* of string s in $\mathcal{T}_S$ is a pair $\phi = (e, h)$, where e is the edge from u to v and h is the offset from u, namely, $h = |s| - |str(u)|$.[3] We extend the str function to locus ϕ, so that $str(\phi) = s$. The *string depth* of locus ϕ is $d(\phi) = |str(\phi)|$. A string P is *recognized* by

[3]In the literature the locus is represented by (u, c, h) where c is the first letter of the label of e. Since our packed c-trie does not maintain a search structure for branches, we represent the locus directly on e.

$\mathcal{T}_S$ iff there is a locus ϕ with $str(\phi) = P$.

We consider the following query and operations on dynamic compact tries.

$\mathsf{LPS}(\phi, P)$: Given a locus in $\mathcal{T}_S$ and a pattern string P, it returns the locus $\hat{\phi}$ of string $str(\phi)Q$ in $\mathcal{T}_S$, where Q is the longest prefix of P for which $str(\phi)Q$ is recognized by $\mathcal{T}_S$. When $\phi = ((\perp, root(\mathcal{T}_S)), 1)$, then the query is known as the *longest prefix search* for the pattern P in the compact trie.

$\mathsf{Insert}(\phi, X)$: Given a locus ϕ in $\mathcal{T}_S$ and a string X, it inserts a new leaf which corresponds to a new string $str(\phi)X \in S$ into the compact trie, from the given locus ϕ. When there is no node at the locus $\hat{\phi} = \mathsf{LPS}(\phi, X)$, then a new node is created at $\hat{\phi}$ as the parent of the leaf. When $\phi = ((\perp, root(\mathcal{T}_S)), 1)$, then this is standard insertion of string X to $\mathcal{T}_S$.

$\mathsf{Delete}(X_i)$: Given a string $X_i \in S$, it deletes the leaf ℓ_i. If the out-degree of the parent v of ℓ_i becomes 1 after the deletion of ℓ_i, then the in-coming and out-going edges of v are merged into a single edge, and v is also deleted.

3.2.2 Dynamic predecessor data structures.

For a dynamic set $I \subseteq [1, n]$ of k integers of $w = \log n$ bits each, *dynamic predecessor data structures* (e.g., [6, 7, 71]) efficiently support predecessor query $\mathsf{Pred}(X) = \max(\{Y \in I \mid Y \leq X\} \cup \{0\})$, successor query $\mathsf{Succ}(X) = \min(\{Y \in I \mid Y \leq X\} \cup \{n+1\})$, and insert/delete operations for I.

Theorem 3 *Let $f(k, n)$ be the time complexity of for predecessor/successor queries and insert/delete operations of an arbitrary dynamic predecessor/successor data structure which occupies $O(k \log n)$ bits of space. Beame and Fich's data structure [6] achieves $f(k, n) = O(\frac{(\log\log k)(\log\log n)}{\log\log\log n})$ worst-case time.*

Theorem 4 *Let $f(k, n)$ be the time complexity of for predecessor/successor queries and insert/delete operations of an arbitrary dynamic predecessor/successor data structure*

which occupies $O(k \log n)$ bits of space. Willard's Y-fast trie [70] achieves $f(k, n) = O(\log \log n)$ expected time.

3.3 Packed dynamic compact tries

This section presents our new dynamic compact tries called the *packed dynamic compact tries* (*packed c-tries*) for a dynamic set $S = \{X_1, \ldots, X_k\}$ of k strings of total length n, which achieves the main result in Theorem 1. In the sequel, a string $X \in \Sigma^*$ is called *short* if $|X| \leq \alpha = \log_\sigma n$, and is called *long* if $|X| > \alpha$.

3.3.1 Micro dynamic compact tries for short strings.

In this subsection, we present our data structure storing short strings. Our input is a dynamic set $S = \{X_1, \ldots, X_k\}$ of k strings of total length n, such that $|X_i| \leq \alpha = \log_\sigma n$ for every $1 \leq i \leq k$. Hence it holds that $k \leq \sigma^\alpha = n$. For simplicity, we assume for now that $|X_i| = \alpha$ for every $1 \leq i \leq k$. The general case where S contains strings shorter than α will be explained later in Remark 1.

The dynamic data structure for short strings, called a *micro c-trie* and denoted $\mathcal{MT}_S$, consists of the following: (i) A dynamic compact trie of height exactly α storing the set S. Let $\mathcal{N}$ be the set of internal nodes, and let $\mathcal{L} = \{\ell_1, \ldots, \ell_k\}$ be the set of k leaves such that ℓ_i corresponds to X_i for $1 \leq i \leq k$. Since every internal node is branching, $|\mathcal{N}| \leq k-1$. Every node v of $\mathcal{MT}_S$ corresponds to the string $str(v)$ of $\log n$ bits. Overall, this compact trie requires $n \log \sigma + O(k \log n)$ bits of space (including S). (ii) A dynamic predecessor/successor data structure $\mathcal{D}$ which stores the set $S = \{X_1, \ldots, X_k\}$ of strings in $O(k \log n)$ bits of space, where each X_i is regarded as a $\log n$-bit integer. $\mathcal{D}$ supports predecessor/successor queries and insert/delete operations in $f(k, n)$ time each. Clearly $\mathcal{MT}_S$ requires $n \log \sigma + O(k \log n)$ bits of total space.

The next lemma shows how to support in $O(1)$ time LCP queries for strings repre-

sented by two given nodes on the dynamic micro c-trie $\mathcal{MT}_S$. This is related to the *labeling scheme* (e.g., see [1]) which assigns a short label to each node so that later, given the labels of two nodes, the label of the LCA of the nodes can be answered in $O(1)$ time. Although the static tree is considered in the labeling scheme, our micro c-trie is dynamic. Also, our algorithm is much simpler than applying the dynamic LCA data structure [20] to our micro c-tries.

Lemma 1 *For any nodes u and v of the dynamic micro c-trie $\mathcal{MT}_S$, we can compute $LCP(str(u), str(v))$ in $O(1)$ time.*

Proof 1 *We pad $str(u)$ and/or $str(v)$ with an arbitrary letter c so they become α long each, namely, let $P = str(u)c^{\alpha-|str(u)|}$ and $Q = str(v)c^{\alpha-|str(v)|}$. We compute the most significant bit (msb) of the XOR of the bit representations of P and Q. Let b the bit position of the msb, and let $z = (b-1)/\log\sigma$. W.l.o.g. assume $|str(u)| \leq |str(v)|$. (1) If $z < str(u)$, then $str(u)[1, z] = LCP(str(u), str(v))$. In this case, there exists a branching node y such that $str(y) = str(u)[1, z]$, and hence $LCP(str(u), str(v)) = str(y)$. (2) If $z \geq str(u)$, then $str(u) = LCP(str(u), str(v))$, and hence $str(u) = LCP(str(u), str(v))$.*

Since each of P and Q is stored in a single machine word, we can compute the XOR of P and Q in $O(1)$ time. The msb can be computed in $O(1)$ time using the technique of Fredman and Willard [35]. This completes the proof. ∎

On micro c-tries, prefix searches and insertion operations can be started not only from the root but from *any* node. This is necessary for online sparse suffix tree construction based on Ukkonen's algorithm [65], since during the suffix link traversal we have to insert new leaves from non-root internal nodes.

Theorem 5 *The micro c-trie $\mathcal{MT}_S$ supports $\mathsf{LPS}(\phi, X)$ queries in $O(f(k, n))$ time.*

Proof 2 *Let P be the prefix of $str(\phi)X$ of length α, i.e., $P = str(\phi)X[1..\alpha - d(phi)]$. The case where P is represented by a leaf is easy, and thus, in what follows we focus on the case where P is not represented by a leaf.*

First, we compute the string depth $d = d(\phi) \in [0, \alpha]$. Observe that $d = \max\{|LCP(P, \mathsf{Pred}(P))|, |LCP(P, \mathsf{Succ}(P))|\}$. Given P, we compute $\mathsf{Pred}(P)$ and $\mathsf{Succ}(P)$ in $O(f(k, n))$ time. Then, we can compute $|LCP(P, \mathsf{Pred}(P))|$ in $O(1)$ time by computing the msb of the XOR of the bit representations of P and $\mathsf{Pred}(P)$, as in Lemma 1. $|LCP(P, \mathsf{Succ}(P))|$ can be computed analogously, and thus, $d = d(\phi)$ can be computed in $O(f(k, n))$ time.

Second, we locate $e = (u, v)$. See also Fig. 3.1. Let $Z = P[1, d]$. Let $LB = Zc_1^{\alpha-|Z|}$ and $UB = Zc_\sigma^{\alpha-|Z|}$ be the lexicographically least and greatest strings of length α with prefix Z, respectively. To locate u in $\mathcal{MT}_S$, we find the leftmost and rightmost leaves X_L and X_R below ϕ by $X_L = \mathsf{Succ}(LB)$ and $X_R = \mathsf{Pred}(UB)$. Then, the longer one of $LCP(X_{L-1}, X_L)$ and $LCP(X_R, X_{R+1})$ corresponds to the origin node u of e, and $LCP(X_L, X_R)$ corresponds to the destination node v of e. These LCPs can be computed in $O(1)$ time by Lemma 1. What remains is how to access the nodes u and v representing these strings. In so doing, let \$ be a special character that does not appear in any strings in S. For each string Y represented by an internal node of $\mathcal{MT}_S$, we pad \$ at the end of Y so its length becomes exactly α, namely, we obtain $Y\$^{\alpha-|Y|}$. We insert this padded string into a dynamic dictionary dedicated only for internal nodes (here we use a predecessor/successor data structure). Now, given a string represented by an internal node, we can access the corresponding node in $O(f(k, n))$ time. Finally we obtain $\phi = ((u, v), d - |str(u)|)$ in overall $O(f(k, n))$ time. ∎

It follows from the proof of Theorem 5 that a dynamic predecessor/successor data structure is enough to support pattern matching queries on our dynamic micro c-tire. This implies that we do not have to store (the triples for) the edge labels in the micro c-trie. This observation is important when we consider delete operations on the set S, as we will see in the next lemma.

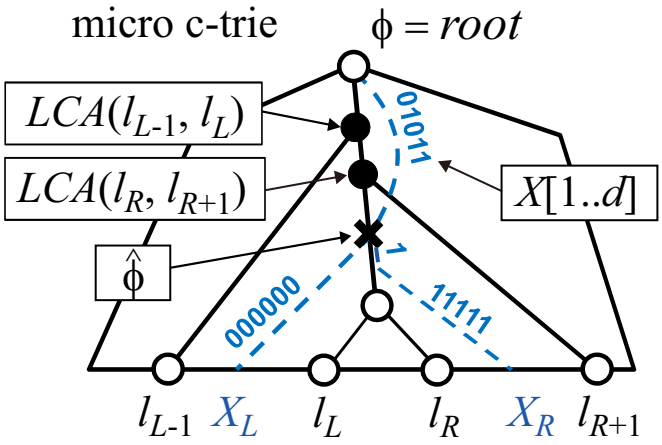

Figure 3.1: Given the initial locus ϕ (which is on the root in this figure) and query pattern $P = 01011010110$, the algorithm of Theorem 5 answers the $\mathsf{LPS}(\phi, P)$ query on the micro c-trie as in this figure. The answer to the query is the locus $\hat{\phi}$ for $P[1..5] = 01011$.

Lemma 2 *The micro c-trie $\mathcal{MT}_S$ supports* $\mathsf{Insert}(\phi, X)$ *and* $\mathsf{Delete}(X)$ *operations in $O(f(k,n))$ time. We assume that $d(\phi) + |X| \leq \alpha$ so that the height of the micro compact trie will always be kept within α.*

Proof 3 *We show how to support* $\mathsf{Insert}(\phi, X)$ *in $O(f(k,n))$ time. Initially $S = \emptyset$, the micro compact trie $\mathcal{MT}_S$ consists only of $root(\mathcal{MT}_S)$, and predecessor/successor dictionary $\mathcal{D}$ contains no elements. When the first string X is inserted to S, then we create a leaf below the root and insert X to $\mathcal{D}$. Suppose that the data structure maintains a string set S with $|S| \geq 1$. To insert a string X from the given locus ϕ, we first conduct the* $\mathsf{LPS}(\phi, X)$ *query of Theorem 5, and let $\hat{\phi} = (e, h)$ be the answer to the query. If $h = |e|$, then we simply insert a new leaf ℓ from the destination node of e. Otherwise, we split e at $\hat{\phi}$ and create a new node v there as the parent of the new leaf, such that $str(v) = str(\hat{\phi})$. The rest is the same as in the former case. After the new leaf is inserted, we insert $str(\phi)X$ to $\mathcal{D}$ in $O(f(k,n))$ time.*

We consider $\mathsf{Delete}(X)$. *Recall that each edge of the micro c-trie does* not *store*

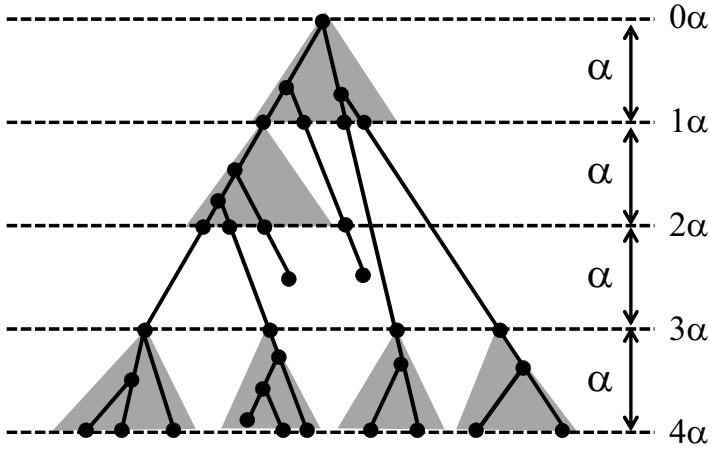

Figure 3.2: Micro-trie decomposition: The packed c-trie is decomposed into a number of micro c-tries (gray rectangles) each of which is of height $\alpha = \log_\sigma n$. Each micro-trie is equipped with a dynamic predecessor/successor data structure.

the triple representing its string label. Thanks to this property, we need not consider updates of the labels of the edges in the path from the root to the deleted leaf (which usually becomes problematic in compact tries). Thus, we can support $\mathsf{Delete}(X)$ *in a similar way to* $\mathsf{Insert}(\phi, X)$*, in* $O(f(k, n))$ *time.*

Remark 1 *When* $d(\phi) + |X| < \alpha$*, then we can support* $\mathsf{Insert}(\phi, X)$ *and* $\mathsf{LPS}(\phi, X)$ *as follows. When inserting* X*, we pad* X *with a special letter* \$ *which does not appear in* S*. Namely, we perform* $\mathsf{Insert}(\phi, X)$ *operation with* $X' = X\$^{\alpha - d(\phi) - |X|}$*. When computing* $\mathsf{LPS}(\phi, X)$*, we pad* X *with another special letter* $\# \neq \$$ *which does not appear in* S*. Namely, we perform* $\mathsf{LPS}(\phi, X'')$ *query with* $X' = X\#^{\alpha - d(\phi) - |X|}$*. This gives us the correct locus for* $\mathsf{LPS}(\phi, X)$*.*

3.3.2 Packed dynamic compact tries for long strings.

In this subsection, we present the *packed dynamic compact trie* (*packed c-trie*) $\mathcal{PT}_S$ for a set S of variable-length strings of length at most $O(2^w) = O(n)$.

3.3.3 Micro trie decomposition.

We decompose $\mathcal{PT}_S$ into a number of micro c-tries. See also Fig. 3.2. Let $h > \alpha$ be the length of the longest string in S. We categorize the nodes of $\mathcal{PT}_S$ into $\lceil h/\alpha \rceil + 1$ levels: We say that a node of $\mathcal{PT}_S$ is at level i ($0 \leq i \leq \lceil h/\alpha \rceil$) iff $|str(v)| \in [i\alpha, (i+1)\alpha - 1]$. The level of a node v is denoted by $level(v)$. A locus ϕ of $\mathcal{PT}_S$ is called a *boundary* iff $d(\phi)$ is a multiple of α. Consider any path from $root(\mathcal{PT}_S)$ to a leaf, and assume that there is no node at some boundary $k\alpha$ on this path. We create an auxiliary node at that boundary on this path, iff there is at least one non-auxiliary (i.e., original) node at level $i-1$ or $i+1$ on this path. Let $\mathcal{BN}$ denote the set of nodes at the boundaries, called the *boundary nodes*. For each boundary node $v \in \mathcal{BN}$, we create a micro compact trie $\mathcal{MT}$ whose root $root(\mathcal{MT})$ is v, internal nodes are all descendants u of v with $level(u) = level(v)$, and leaves are all boundary descendants ℓ of v with $level(\ell) = level(v) + 1$. Notice that each boundary node is the root of a micro c-trie at its level and is also a leaf of a micro c-trie at the previous level. An edge is said to be a *long edge* iff its label is at least α long. We store the label of each long edge by a triple of integers. Recall that, on the other hand, we do not store (encodings) of the edge labels in the micro c-tries.

Lemma 3 *The packed c-trie $\mathcal{PT}_S$ for a prefix-free set S of k strings requires $n \log \sigma + O(k \log n)$ bits of space.*

Proof 4 *Firstly, we show the number of auxiliary boundary nodes in $\mathcal{PT}_S$. At most 2 auxiliary boundary nodes are created on each* original *edge of $\mathcal{PT}_S$. Since there are at most $2k-2$ original edges, the total number of auxiliary boundary nodes is at most $4k-4$.*

Since there are at most $2k-1$ original nodes in $\mathcal{PT}_S$, the total number of nodes in $\mathcal{PT}_S$ is at most $6k-5$. Clearly, the total number of short *strings of length at most α maintained by the micro c-tries is no more than the number of all nodes in $\mathcal{PT}_S$. The*

number of long edges in $\mathcal{PT}_S$ is no more than the number of its nodes. Overall, the total space of $\mathcal{PT}_S$ is $n \log \sigma + O(k \log n)$ bits. ∎

For any locus ϕ on $\mathcal{PT}_S$, $ld(\phi)$ denotes the local string depth of ϕ in the micro c-trie $\mathcal{MT}$ that contains ϕ. Namely, if $root(\mathcal{MT}) = v$, the parent of u in $\mathcal{PT}_S$ is u, and $e = (u, v)$, then $ld(\phi) = d(\phi) - d((e, |e|))$. Prefix search queries and insert/delete operations can be supported by our packed c-trie, as follows.

Lemma 4 *The packed c-trie $\mathcal{PT}_S$ supports $\mathsf{LPS}(\phi, P)$ query in $O(\frac{m}{\alpha} f(k, n))$ worst-case time, where $m = |P| > \alpha$.*

Proof 5 *If $m + ld(\phi) \leq \alpha$, the bound immediately follows from Theorem 5. Assume $m + ld(\phi) > \alpha$, and let $q = \alpha - ld(\phi) + 1$. We factorize P into $h + 1$ blocks as $p_0 = P[1, q - 1]$, $p_1 = P[q, q + \alpha - 1]$, $\ldots$, $p_{h-1} = P[q + (h - 1)\alpha, q + h\alpha - 1]$, and $p_h = P[q + h\alpha, m]$, where $1 \leq |p_0| \leq \alpha$, $|p_i| = \alpha$ for $1 \leq i \leq h - 1$, and $1 \leq |p_h| \leq \alpha$. Each block can be computed in $O(1)$ time by standard bit operations. If there is a mismatch in p_0, we are done. Otherwise, for each i in increasing order from 1 to h, we perform $\mathsf{LPS}(\gamma, p_i)$ query from the root γ of the corresponding micro c-trie at each level of the corresponding path starting from ϕ. This continues until we find either the first mismatch for some i or complete matches for all i's. Each LPS query with each micro c-trie takes $O(f(k, n))$ time by Theorem 5. Since $h = O(\frac{m}{\alpha})$, it takes $O(\frac{m}{\alpha} f(k, n))$ total time.* ∎

Lemma 5 *The packed c-trie $\mathcal{PT}_S$ supports $\mathsf{Insert}(\phi, X)$ and $\mathsf{Delete}(X_i)$ operations in $O(\frac{m}{\alpha} f(k, n))$ worst-case time, where $m = |X| > \alpha$.*

Proof 6 $\mathsf{Insert}(\phi, X)$*: we first perform $\mathsf{LPS}(\phi, X)$ in $O(\frac{m}{\alpha} f(k, n))$ time (Lemma 4). Let $x_0, \ldots, x_h$ be the factorization of X w.r.t. ϕ, and let x_j be the block of the factorization containing the first mismatch. Then, we conduct $\mathsf{Insert}(\gamma, x_j)$ operation on the corresponding micro c-trie, where γ is its root. It takes $O(f(k, n))$ time (Lemma 2).*

If $j = h$ (x_j is the last block in the factorization of X), then we are done. Otherwise, we create a new edge with label $x'_j x_{j+1} \cdots x_k$, where x'_j is the suffix of X_j which begins at the mismatched position, leading to the new leaf ℓ. We create a new boundary node if necessary. These operations take $O(1)$ time each. Hence, $\mathsf{Insert}(\phi, X)$ *takes $O(\frac{m}{\alpha} f(k,n))$ total time.*

$\mathsf{Delete}(X_i)$*: Let Q be the path from the root r of $\mathcal{PT}_S$ to leaf ℓ_i. If ℓ_i is a child of the root of $\mathcal{PT}_S$, then we simply delete the single edge in Q. Otherwise, for each sub-path of Q that belongs to a micro c-trie, we perform* Delete *operation of Lemma 2 in this micro c-trie. Since the path Q spans at most $\frac{m}{\alpha}$ micro c-tries, the delete operations on these micro c-tries take $O(\frac{m}{\alpha} f(k,n))$ total time. For each long edge in Q whose label refers to X_i, let $\langle i, a, b\rangle$ be the triple representing the label. We replace the triple with $\langle i', a', b'\rangle$, where $X_{i'}$ is the predecessor of X_i in S and $X_{i'}[a'..b'] = X[a..b]$ (if X_i does not have a predecessor, then we can use the successor of S instead). We can find $X_{i'}$ as follows. First, we compute $\phi = \mathsf{LPS}(r, X_i) = LCA(\ell_{i'}, \ell_i)$. Then, we can find $\ell_{i'}$ by traversing the right-most path from ϕ that is to the left of the sub-path of Q from ϕ to ℓ_i. This can be done in $O(\frac{m}{\alpha} f(k,n))$ time. The positions a' and b' in $X_{i'}$ can be computed by simple arithmetics, since we know the total length of the labels in the path from ϕ to $\ell_{i'}$. Since the path Q contains less than $\frac{m}{\alpha}$ long edges, the triples for all long edges in Q can be updated in $O(\frac{m}{\alpha})$ time.* ∎

3.3.4 Speeding-up with hashing.

By augmenting each micro c-trie with a hash table storing the short strings, we achieve a good expected performance, as follows:

Lemma 6 *The packed c-trie $\mathcal{PT}_S$ augmented with hashing supports* $\mathsf{LPS}(\phi, X)$ *query,* $\mathsf{Insert}(\phi, X)$ *and* $\mathsf{Delete}(X)$ *operations in $O(\frac{m}{\alpha} + f(k,n))$ expected time.*

Proof 7 *Let $\mathcal{MT}$ be any micro c-trie in the packed c-trie $\mathcal{PT}_S$, and M the set of*

strings maintained by $\mathcal{MT}$ each being of length at most α. We store all strings of M in a hash table associated to $\mathcal{MT}$, which supports look-ups, insertions and deletions in $O(1)$ expected time.

Let $x_0, \ldots, x_h$ be the factorization of X w.r.t. ϕ. To perform $\mathsf{LPS}(\phi, X)$, we ask if $str(\phi)x_0$ is in the hash table of the corresponding micro c-trie. If the answer is no, the first mismatch occurs in x_0, and the rest is the same as in Lemma 4. If the answer is yes, then for each i from 1 to h in increasing order, we ask if x_i is in the hash table of the corresponding micro c-trie, until we receive the first no with some i or we receive yes for all i's. In the latter case, we are done. In the former case, we perform LPS *query with x_i from the root of the corresponding micro c-trie. Since we perform at most one* LPS *query and $O(\frac{m}{\alpha})$ look-ups for hash tables, it takes $O(\frac{m}{\alpha} + f(k, n))$ expected time. $O(\frac{m}{\alpha} + f(k, n))$ expected time bounds for $\mathsf{Insert}(\phi, X)$ and $\mathsf{Delete}(X)$ immediately follow from the above arguments.* ∎

3.4 Applications to online string processing

Sparse suffix trees. The *suffix tree* [68] of a string T of length n is a compact trie which stores all n suffixes of T. A *sparse suffix tree* for a set $K \subseteq [1, n]$ of *sampled positions* of T is a compact trie which stores only the subset $S = \{T[i..n] \mid i \in K\}$ of the suffixes of T beginning at the sampled positions in K. It is known that if the set K of sampled positions satisfy some properties (e.g., every r positions for some fixed $r > 1$ or the positions immediately after the word delimiters), the sparse suffix tree can be constructed in an online manner in $O(n \log \sigma)$ time and $n \log \sigma + O(n \log n)$ bits of space [45, 47, 64].

Packed c-tries can speed up online construction and pattern matching for these sparse suffix trees: Here each input string X to Insert is given as a pair (i, j) of positions in T s.t. $X = T[i..j]$. As Lemma 7 states, Insert operation in such a case can be

processed more quickly than in Lemma 4.

Lemma 7 *Given a pair (i, j) of positions in T s.t. $X = T[i..j]$, we can support* $\mathsf{Insert}(\phi, X)$ *in $O(\frac{q}{\alpha} f(k, n))$ worst-case time or $O(\frac{q}{\alpha} + f(k, n))$ expected time, where q is the length of the longest prefix of X that can be spelled out from ϕ.*

Theorem 6 *Using packed c-tries, we can construct in an online manner the sparse suffix trees of [45, 47, 64] for a given text T of length n in $O((\frac{n}{\alpha} + k) f(k, n))$ worst-case time or in $O(\frac{n}{\alpha} + k f(k, n))$ expected time with $n \log \sigma + O(k \log n)$ bits of space, where k is the number of sampled positions. At any moment during the construction, pattern matching queries take $O(\frac{m}{\alpha} f(k, n))$ worst-case time or in $O(\frac{m}{\alpha} + f(k, n))$ expected time, where m is the the pattern length.*

Proof 8 *We explain how we can buid the sparse suffix trees of [47] efficiently. For an integer parameter $r > 1$, Kärkkäinen and Ukkonen's algorithm (KU-algorithm, in short) [47] constructs the r-*evenly sparse suffix tree *of the input string T. KU-algorithm differs from Ukkonen's online suffix tree construction algorithm in that KU-algorithm uses r-*letter suffix links*, such that the suffix link of each node v is a pointer to the node u such that $str(u) = str(v)[r+1..|str(v)|]$, but otherwise is the same as Ukkonen's algorithm. This results in a compact trie which stores the evenly-spaced $\lceil n/r \rceil$ suffixes $T[1, n]$, $T[1 + r, n]$, $\ldots$, $T[\lceil n/r \rceil, n]$ of T.*

KU-algorithm scans the input string T from left to right, and when the algorithm processes the ith letter of T, the r-evenly sparse suffix tree of $T[1, i]$ is maintained. This is done by inserting the leaves into the current compact trie in increasing order of the positions the leaves correspond to. Assume that while processing the ith letter of T, the algorithm has just inserted the jth leaf ℓ_j for sampled position $1 + (j - 1)r$ of T. If the suffix $T[1 + jr, i]$ of $T[1, i]$ is not recognized by the current compact trie, then the algorithm inserts the $(j + 1)$th leaf ℓ_{j+1} for the next sampled position $1 + jr$. This can be done as follows: For any node v, let $sl_r(v)$ denotes the r-letter suffix link of

v. Let v_j be the nearest ancestor of ℓ_j for which $sl_r(v_j)$ is already defined (v_j is either $parent(\ell_j)$ or $parent(parent(\ell_j))$). We follow the suffix link and let $u_{j+1} = sl_r(v_j)$. Let ϕ_{j+1} be the locus of $str(u_{j+1})$, namely $\phi_{j+1} = (e, |e|)$ with $e = (parent(u_{j+1}), u_{j+1})$. Let $X_{j+1} = T[i-h+1, i]$, where $h = |T[j+1, i]| - |str(\phi_{j+1})| = i - j - |str(\phi_{j+1})|$. The leaf ℓ_{j+1} can be added to the compact trie by inserting the string X_{j+1} from the locus ϕ_{j+1}.

We apply our micro-trie decomposition to the sparse suffix tree, and use our techniques in Section 5.3 and in Lemma 7. Then, the total time complexity to construct the r-evenly sparse suffix tree of T is propotinal to the amount of work of the Insert *operations of Lemma 7 for all leaves. For each $1 \leq j \leq k$ let q_j be the length of the longest prefix of X_j that can be spelled out from ϕ_j. Now we estimate $\sum_{j=1}^{k} \frac{q_j}{\alpha}$. Each time we traverse an r-letter skipping suffix link, the string depth decreases by r. Since $k = \lceil n/r \rceil$ and we traverse r-letter suffix links exactly $k-1$ times, we can colcude that $\sum_{j=1}^{k} q_j = O(n)$, which implies that $\sum_{j=1}^{k} \frac{q_j}{\alpha} = O(n/\alpha)$. Since we perform* Insert *operations exactly k times, the r-evenly sparse suffix tree can be constructed in $O((\frac{n}{\alpha} + k)f(k, n))$ worst-case time or in $O(\frac{n}{\alpha} + kf(k, n))$ expected time.*

The bounds for word suffix trees of Inenaga and Takeda [45] and those of suffix trees on variable-length codes of Uemura and Arimura [64] can be obtained similarly. ∎

LZ-Double factorization. *LZ-Double factorization* (*LZDF*) [39] is a generalization of Lempel-Ziv 78 factorization [73]. The ith factor $g_i = g_{i_1} g_{i_2}$ of the LZDF of a string T of length n is the concatenation of previous factors g_{i_1} and g_{i_2} s.t. g_{i_1} is the longest prefix of $T[1 + \sum_{j=1}^{i-1} |g_j|, n]$ that is a previous factor (one of $\{g_1, \ldots, g_{i-1}\} \cup \Sigma$), and g_{i_2} is the longest prefix of $T[1 + |g_{i_1}| + \sum_{j=1}^{i-1} |g_j|, n]$ that is a previous factor. Goto et al. [39] proposed a Patricia-tree based algorithm which computes the LZDF of a given string T of length n in $O(k(M + \min\{k, M\} \log \sigma))$ worst-case time[4] with $O(k \log n) = O(n \log \sigma)$

[4]Since $kM \geq n$ always hods, the n term is hidden in the time complexity.

bits of space[5], where k is the number of factors and M is the length of the longest factor. With packed c-tries, we can achieve a good expected performance:

Theorem 7 *Using our packed c-trie, we can compute the LZDF of string T in $O(k(\frac{M}{\alpha} + f(k,n)))$ expected time with $O(n \log \sigma)$ bits of space.*

Proof 9 *Suppose we have computed the first $j-1$ factors $g_1, \ldots, g_{j-1}$ and we are now computing the jth factor g_j. We store the previous factors $g_1, \ldots, g_{j-1}$ in our packed c-trie. In addition, there is no leaf or branching node which represents some previous factor g_i ($1 \leq i < j$), then we add an internal non-branching node for g_i into the packed trie. We mark only and all nodes which represent previous factors. To compute the jth factor $g_j = g_{j_1} g_{j_2}$, we perform $\mathsf{LPS}(r, T_j)$ query where r is the locus for the root and $T_j = T[1 + \sum_i^{j-1} |g_i|, n]$. Let $\hat{\phi}$ be the answer to the query. Note that $\hat{\phi}$ can be deeper than the locus for g_{j_1}, but it is always in the subtree rooted at g_{j_1}. Hence, the nearest marked ancestor (NMA) of $\hat{\phi}$ is g_{j_1}. We can compute g_{j_2} similarly. After we computed g_j, we perform $\mathsf{Insert}(r, g_j)$ operation and then mark the node which represents g_j.*

The depth of the locus $\hat{\phi}$ is bounded by the length M of the longest factor. Hence we can reach the locus $\hat{\phi}$ in $O(\frac{M}{\alpha} + f(k,n))$ expected time using our packed c-trie. We repeat the above procedure k times. Using the semi-dynamic NMA data structure of Westbrook [69] that supports NMA queries, inserting new nodes, and marking unmarked nodes in amortized $O(1)$ time each, we obtain the desired bound. ∎

3.5 Preliminary experiments

This section shows some preliminary experimental results which compare our implementations of packed c-tries against that of the classical c-trie (Patricia tree). Table 3.1 shows the datasets and their statistics used in our experiments, where the first

[5]Since all the factors of the LZDF are distinct, $k = O(\frac{n}{\log_\sigma n})$ holds [73].

Table 3.1: Description of the datasets we used in our experiments.

Dataset	Original alhpabet size	Actual alphabet size	Total size (bytes)	Number of strings	Ave. string length (bits)
DNA	4	2	52,428,800	337	1,244,600.59
DBLP	128	2	52,428,800	3,229,589	129.87
english	128	2	52,428,800	9,400,185	44.62
pitches	128	2	52,428,800	93,354	4,492.90
proteins	20	2	52,428,800	186,914	2,243.98
sources	128	2	52,428,800	5,998,228	69.93
urls	128	2	52,010,031	707,658	587.97
jawiki	$\geq 2^{16}$	2	30,414,297	1,643,827	148.02

six datasets are from Pizza&Chili Corpus[6], the seventh consists of URLs in `uk` domain[7], and the eighth consists of all titles from Japanese Wikipedia[8]. The datasets were treated as binary strings.

We tested three implementations of c-tries by the authors: an implementation CT of classical c-tries, and two simplified implementations $\mathsf{PCT}_{\mathsf{xor}}$ and $\mathsf{PCT}_{\mathsf{hash}}$ of our packed c-tries in Section 5.3 as a proof-of-concept versions. CT uses *unordered_map* in the C++/STL library to maintain the branching out-going edges of its nodes. For our implementations of packed c-tries, we set $\alpha = 32$. The first implementation $\mathsf{PCT}_{\mathsf{xor}}$ only uses the XOR-based technique of Theorem 4 to quickly process long edges, while branching out-going edges are processed as in CT. The second implementation $\mathsf{PCT}_{\mathsf{hash}}$ is a simplified version of our packed c-tries of Lemma 6 using XOR and hashing. Each micro c-trie in $\mathsf{PCT}_{\mathsf{hash}}$ is equipped with a hash table for α-bit integers. We again used unordered_map in the C++/STL library for hash tables on micro c-tries. For simplicity, each micro c-trie is not equipped with a predecessor/successor data structure.

We compiled all programs with gcc 4.9.3 using -O3 option, and ran all experiments on a PC (2.8GHz Intel Core i7 processor, register size 64 bits, 16GB of memory)

[6]Pizza&Chili Corpus, http://pizzachili.dcc.uchile.cl

[7]Laboratory for webalgorithmics, uk-2005.urls.gz, http://law.di.unimi.it/datasets.php

[8]jawiki, https://dumps.wikimedia.org/jawiki/

Table 3.2: Summary of our experimental results.

Dataset	Tree size (# of nodes)			Const. time (msec)			Query time (msec)		
	CT	$\mathsf{PCT}_{\mathsf{xor}}$	$\mathsf{PCT}_{\mathsf{hash}}$	CT	$\mathsf{PCT}_{\mathsf{xor}}$	$\mathsf{PCT}_{\mathsf{hash}}$	CT	$\mathsf{PCT}_{\mathsf{xor}}$	$\mathsf{PCT}_{\mathsf{hash}}$
DNA	674	674	985	**14,494**	15,270	18,596	6,690	7,381	**5,342**
DBLP	1,059,656	1,059,656	1,204,651	16,662	16,987	**14,139**	8,083	8,905	**7,209**
english	448,379	448,379	532,750	17,496	**16,944**	18,197	**9,127**	9,916	10,452
pitches	86,205	86,205	121,943	18,816	16,571	**16,520**	7,022	9,009	**6,053**
proteins	310,392	310,392	437,768	17,957	**15,733**	18,673	8,511	8,851	**6,749**
sources	1,314,571	1,314,571	1,616,872	17,398	**15,929**	16,892	8,111	8,444	**7,852**
urls	1,341,200	1,341,200	1,357,730	14,038	**13,422**	13,585	6,939	6,903	**5,918**
jawiki	2,365,821	2,365,821	3,043,817	9,440	**9,116**	10,107	4,477	4,661	**3,962**

running on MacOS X 10.10.5, where consecutive $\alpha = 32$ bits of strings were packed into a machine word.

In Table 3.2, we show our experimental results. First, we consider the first groups of columns for the tree sizes. We observe that the number of nodes of $\mathsf{PCT}_{\mathsf{hash}}$ increases from both of CT and $\mathsf{PCT}_{\mathsf{xor}}$. The gain varies from 101.3% on urls to 146.1% on DNA. This comes from the addition of boundary nodes. Next, we consider the second groups of columns for the construction times. We observe that $\mathsf{PCT}_{\mathsf{xor}}$ is slightly faster than the classical CT in most case. The construction time of $\mathsf{PCT}_{\mathsf{hash}}$ is slightly faster against CT for DBLP, pitches, sources and urls, and slower for DNA, english, proteins and jawiki. Yet, the construction time of $\mathsf{PCT}_{\mathsf{hash}}$ per node is faster than CT for all datasets. We, however, do not observe clear advantage of $\mathsf{PCT}_{\mathsf{hash}}$ over $\mathsf{PCT}_{\mathsf{xor}}$. We guess that the inconsistency is due to the balance of utility and the overhead for creating the boundary nodes. Finally, we consider the third groups of columns for query times. In these experiments, we used all strings from the dataset as query patterns, and searched them on each c-trie. The table shows the total times for all the pattern searches. Among all the datasets except english, $\mathsf{PCT}_{\mathsf{hash}}$ is clearly faster than CT, where the former achieved 5% to 20% speed-up over the latter. This indicates that $\mathsf{PCT}_{\mathsf{hash}}$ is superior to the classic c-tries in prefix searches.

3.6 Conclusions of Chapter 3

In this chapter, we presented a faster version of compact tries, called the *packed compact trie* (*packed c-trie*), which stores a set S of k strings of total length n in $n \log \sigma + O(k \log n)$ bits of space, where σ is the size of an alphabet. The packed c-trie supports pattern matching and insert/delete operations in $O(\frac{m}{\alpha} f(k, n))$ worst-case time and in $O(\frac{m}{\alpha} + f(k, n))$ expected time, where m is the length of a pattern and $f(k, n)$ is a time complexity of a predecessor dictionary. We also showed applications of our packed c-tries. One of applications is faster construction of sparse suffix trees. Another one is speeding-up LZD factorization.

Chapter 4

Fully-online Construction of Suffix trees for Multiple Texts

In this chapter, we consider the construction of the *suffix tree* and the *directed acyclic word graph (DAWG)* indexing data structures for a collection of texts $\mathcal{T}$, where a new symbol may be appended to any text in $\mathcal{T} = \{T_1, \ldots, T_K\}$ *at any time*. This *fully-online* scenario, that arises when dynamically indexing multi-sensor data, is a natural generalization of the long solved *semi-online* problem, where texts $T_1, \ldots, T_{k-1}$ are permanently fixed before the next text T_k is processed. We present fully-online algorithms that constructs the *suffix tree and the DAWG* for $\mathcal{T}$ in $O(n \log \sigma)$ time and $O(n)$ space, where n is the total lengths of the strings in $\mathcal{T}$ and σ is their alphabet size. The standard explicit representation of the suffix tree and the DAWG edges must be relaxed in the fully-online scenario, since too many updates might be required, and instead, we provide access to the frequently updated suffix tree leaf edge labels and the DAWG re-directable edges via auxiliary data structures, in $O(\log \sigma)$ time.

4.1 Background

Text indexing is a fundamental problem in computer science, which plays important roles in many applications including text retrieval, molecular biology, signal processing, and sensor data analysis. In this chapter, we focus on indexing a collection of multiple texts, so that subsequent pattern matching queries can be answered quickly. In particular, we study online indexing for a collection $\mathcal{T}$ of multiple texts, where a new character can be appended to each text at *any* time. Such fully-online indexing for multiple growing texts has potential applications to continuous processing of data streams, where a number of symbolic events or data items are produced from multiple, rapid, time-varying, and unbounded data streams [4, 48]. For example, motif mining system tries to discover characteristic or interesting collective behaviors, such as frequent path or anomalies, from data streams generated by a collection of moving objects or sensors [48, 67].

Many of the existing suffix tree and DAWG construction algorithms [14, 15, 22, 25, 42, 65, 68] for a single text also work within the same $O(n \log \sigma)$ time and $O(n)$ space bounds for a collection of growing texts in the *semi-online* setting, where only the last inserted text can be extended. However, special attention is needed in the *fully-online* setting. When using a direct explicit representation of the DAWG edges, up to $\Theta(n \min(K, \sqrt{n}))$ or $\Theta(n^{1.5})$ DAWG edge re-directions may be required, while the open-ended suffix tree leaf edge label representation, the cornerstone of Ukkonen's [65] on-line suffix tree algorithm, may have to update the association between the numerous suffix tree leaf edge labels and the various texts up to $\Theta(n^2)$ times. Thus, if we wish to stay within the $O(n \log \sigma)$ time bounds in the *fully-online* setting, the DAWG edges and the suffix tree leaf edge labels cannot be directly explicitly maintained.

We propose how the DAWG and the suffix tree can be incrementally constructed for a fully-online text collection. First, we observe that Blumer et al.'s construction [15] for DAWGs and Weiner's right-to-left construction [68] for suffix trees can readily be

adapted to solve this problem. Hence, at any moment during the fully-online growth of the texts, we can find all *occ* occurrences of a given pattern of length m in the current text collection in $O(m \log \sigma + occ)$ time.

Our next goal is to extend Ukkonen's construction [65] to fully-online left-to-right construction of suffix trees for multiple texts. A motivation of this goal is that a growing suffix tree can be enhanced with powerful semi-dynamic tree data structures such as those for *nearest marked ancestor (NMA) queries* [69], *lowest common ancestor (LCA) queries* [20], and *level ancestor (LA) queries* [2]. Note that these data structures cannot be applied to DAWGs, and that the same query results cannot be obtained on the suffix tree maintained in a Weiner-like right-to-left online manner since the suffix tree obtained in this manner inherently indexes the *reversed* texts in the collection. However, it turns out that this goal is a big algorithmic challenge, because: (A) In Ukkonen's algorithm, a pointer called the *active point* keeps track of the insertion points of suffixes in decreasing order of length. The efficiency of Ukkonen's algorithm is due to the monotonicity of the tracking path of the active point. However, unfortunately this monotonicity does not hold in the fully-online setting for multiple texts. (B) Due to the non-monotonicity mentioned above, Ukkonen's technique to amortize the cost to track the suffix insertion points does not work in our case. (C) Ukkonen's "open edge" technique to maintain the leaves does not work in our case, either. In Section 4.4 we will explain in more details why and how these problems arise in our fully-online setting. In this chapter, we present a number of new novel techniques to overcome all the difficulties above. As a final result, we propose the first *optimal* $O(n \log \sigma)$-time $O(n)$-space fully-online left-to-right construction algorithm for a suffix tree of multiple texts over a general ordered alphabet of size σ, where n is the final total length of the texts.

4.1.1 Related work

We note that we can obtain fully-online text index for multiple texts using existing more general dynamic text indices as follows. To use the index of Ferragina and Grossi [29] which permits character-wise updates, we build a text $\$_1 \cdots \$_K$ which initially consists only of K delimiters. Then, appending a character a to the kth text in the collection reduces to prepending a to the kth delimiter $\$_k$. Using this approach, the index of Ferragina and Grossi [29] takes $O(n \log n)$ total time to be constructed, requires $O(n \log n)$ space, and allows pattern matching in $O(m + \log n + n \log m + occ)$ time. Using the compressed index for a dynamic text collection of Chan et al. [17], we can append a new character a to the kth text T_k by removing T_k and then adding $T_k a$ in $O(|T_k|)$ time. This yields a fully-online index with $O(n^2 \log n)$ construction time and $O(n)$ bits of space (or $O(n/\log n)$ words of space assuming $\Theta(\log n)$-bit machine word), supporting pattern matching in $O(m \log n + occ \log^2 n)$ time.

4.2 Preliminaries

4.2.1 Suffix trees and DAWGs for multiple texts

The suffix trie for a text collection $\mathcal{T} = \{T_1, \ldots, T_K\}$, denoted $\mathit{STrie}(\mathcal{T})$, is a trie which represents $\mathit{Suffix}(\mathcal{T})$. The size of $\mathit{STrie}(\mathcal{T})$ is $O(N^2)$, where N is the total length of texts in $\mathcal{T}$. We identify each node v of $\mathit{STrie}(\mathcal{T})$ with the string that v represents. A substring x of a text in $\mathcal{T}$ is said to be *branching* in $\mathcal{T}$, if there exist two distinct characters $a, b \in \Sigma$ such that both xa and xb are substrings of some texts in $\mathcal{T}$. Clearly, node x of $\mathit{STrie}(\mathcal{T})$ is branching iff x is branching in $\mathcal{T}$. For each node av of $\mathit{STrie}(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, let $\mathit{slink}(av) = v$. This auxiliary edge $\mathit{slink}(av) = v$ from av to v is called a *suffix link*.

The *suffix tree* [68] for a text collection $\mathcal{T}$, denoted $\mathit{STree}(\mathcal{T})$, is a "compacted

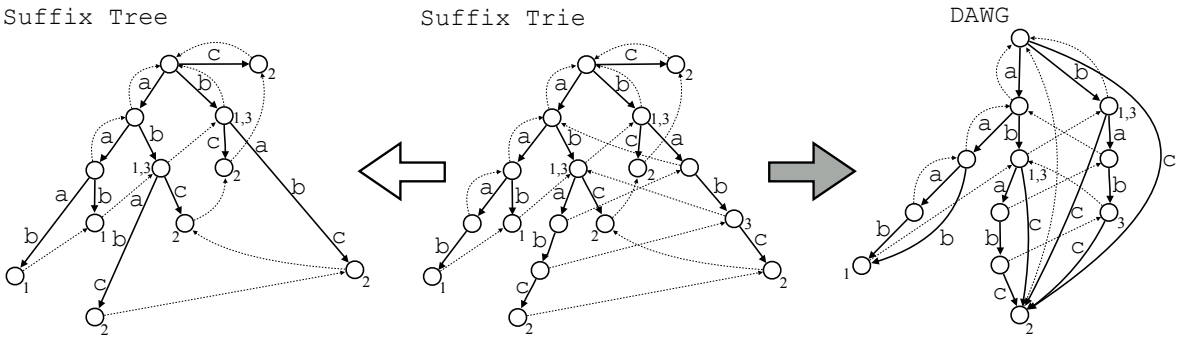

Figure 4.1: Illustration for $STrie(\mathcal{T})$, $STree(\mathcal{T})$, and $DAWG(\mathcal{T})$ with $\mathcal{T} = \{T_1 = \texttt{aaab}, T_2 = \texttt{ababc}, T_3 = \texttt{bab}\}$. The solid arrows and broken arrows represent the edges and the suffix links of each data structure, respectively. The number k ($k = 1, 2, 3$) beside each node indicates that the node represents a suffix of T_k. The nodes $[\texttt{ab}]_{\mathcal{T}}$ and $[\texttt{b}]_{\mathcal{T}}$ are separated in $DAWG(\mathcal{T})$ since the node `bab` in $STrie(\mathcal{T})$ is represents a suffix of T_3, while the node `abab` does not (see also the subtrees rooted at nodes `ab` and `b` in $STrie(\mathcal{T})$).

trie" which represents $Suffix(\mathcal{T})$. $STree(\mathcal{T})$ is obtained by compacting every path of $STrie(\mathcal{T})$ which consists of non-branching internal nodes (see Fig. 4.1). Since every internal node of $STree(\mathcal{T})$ is branching, and since there are at most N leaves in $STree(\mathcal{T})$, the numbers of edges and nodes are $O(N)$. The edge labels of $STree(\mathcal{T})$ are non-empty substrings of some text in $\mathcal{T}$. By representing each edge label x with a triple $\langle k, i, j\rangle$ of integers s.t. $x = T_k[i..j]$, $STree(\mathcal{T})$ can be stored with $O(N)$ space. We say that any branching (resp. non-branching) substring of $\mathcal{T}$ is an *explicit node* (resp. *implicit node*) of $STree(\mathcal{T})$. An implicit node x is represented by a triple (v, a, ℓ), called a *reference* to x, such that v is an explicit ancestor of x, a is the first character of the path from v to x, and ℓ is the length of the path from v to x. A reference (v, a, ℓ) to node x is called *canonical* if v is the lowest explicit ancestor of x. For each node av of $STree(\mathcal{T})$ with $a \in \Sigma$ and $v \in \Sigma^*$, let $slink(av) = v$.

The *directed acyclic word graph* [14, 15] of a text collection $\mathcal{T}$, denoted $DAWG(\mathcal{T})$, is a smallest DAG which represents $Suffix(\mathcal{T})$. $DAWG(\mathcal{T})$ is obtained by merging identical subtrees of $STrie(\mathcal{T})$ connected by the suffix links (see Fig. 4.1). Hence, the

label of every edge of $DAWG(\mathcal{T})$ is a single character. The numbers of nodes and edges of $DAWG(\mathcal{T})$ are $O(N)$ [15], and hence $DAWG(\mathcal{T})$ can be stored with $O(N)$ space. $DAWG(\mathcal{T})$ can be defined formally as follows: For any string x, let $Epos_{\mathcal{T}}(x)$ be the set of ending positions of x in the texts in $\mathcal{T}$, i.e., $Epos_{\mathcal{T}}(x) = \{(k, j) \mid x = T_k[j - |x| + 1..j], 1 \leq j \leq |T_k|, 1 \leq k \leq K\}$. Consider an equivalence relation $\equiv_{\mathcal{T}}$ on substrings x, y of texts in $\mathcal{T}$ such that $x \equiv_{\mathcal{T}} y$ iff $Epos_{\mathcal{T}}(x) = Epos_{\mathcal{T}}(y)$. For any substring x of texts of $\mathcal{T}$, let $[x]_{\mathcal{T}}$ denote the equivalence class w.r.t. $\equiv_{\mathcal{T}}$. There is a one-to-one correspondence between each node v of $DAWG(\mathcal{T})$ and each equivalence class $[x]_{\mathcal{T}}$, and hence we will identify each node v of $DAWG(\mathcal{T})$ with its corresponding equivalence class $[x]_{\mathcal{T}}$. Let $long([x]_{\mathcal{T}})$ denote the longest member of $[x]_{\mathcal{T}}$. By the definition of equivalence classes, $long([x]_{\mathcal{T}})$ is unique for each $[x]_{\mathcal{T}}$ and every member of $[x]_{\mathcal{T}}$ is a suffix of $long([x]_{\mathcal{T}})$. If x, xa are substrings of some text in $\mathcal{T}$ with $x \in \Sigma^*$ and $a \in \Sigma$, then there exists an edge labeled with character $a \in \Sigma$ from node $[x]_{\mathcal{T}}$ to node $[xa]_{\mathcal{T}}$. This edge is called *primary* if $|long([x]_{\mathcal{T}})| + 1 = |long([xa]_{\mathcal{T}})|$, and is called *secondary* otherwise. For each node $[x]_{\mathcal{T}}$ of $DAWG(\mathcal{T})$ with $|x| \geq 1$, let $slink([x]_{\mathcal{T}}) = y$, where y is the longest suffix of $long([x]_{\mathcal{T}})$ which does not belong to $[x]_{\mathcal{T}}$. In the example of Fig. 4.1, $[\texttt{aaab}]_{\mathcal{T}} = \{\texttt{aaab}, \texttt{aab}\}$. The edge labeled with $\texttt{b}$ from node $[\texttt{aaa}]_{\mathcal{T}}$ to node $[\texttt{aaab}]_{\mathcal{T}}$ is primary, while the edge labeled with $\texttt{b}$ from $[\texttt{aa}]_{\mathcal{T}}$ to node $[\texttt{aaab}]_{\mathcal{T}}$ is secondary. $slink([\texttt{aaab}]_{\mathcal{T}}) = [\texttt{ab}]_{\mathcal{T}}$.

The following fact follows from the definition of branching substrings:

Fact 1 *For any substring x of texts in $\mathcal{T}$, node x is branching (explicit) in $STree(\mathcal{T})$ iff node $[x]_{\mathcal{T}}$ is branching in $DAWG(\mathcal{T})$.*

4.2.2 Fully-online text collection

We consider a collection $\{T_1, \ldots, T_K\}$ of K growing texts, where each text T_k ($1 \leq k \leq K$) is initially the empty string ε. Given a pair (k, a) of a text id k and a character $a \in \Sigma$ which we call an *update operator*, the character a is appended to the k-th text of the col-

lection. For a sequence U of update operators, let $U[1..i]$ denote the sequence of the first i update operators in U with $0 \leq i \leq |U|$. Also, for $0 \leq i \leq |U|$ let $\mathcal{T}_{U[1..i]}$ denote the collection of texts which have been updated according to the first i update operators of U. For instance, consider a text collection of three texts which grow according to the following sequence $U = (1, \mathtt{a}), (2, \mathtt{b}), (2, \mathtt{a}), (3, \mathtt{a}), (1, \mathtt{a}), (3, \mathtt{c}), (3, \mathtt{b}), (2, \mathtt{b}), (1, \mathtt{a}), (1, \mathtt{b}), (3, \mathtt{c}), (3, \mathtt{b}), (1, \mathtt{c})$ $(2, \mathtt{c})$ of 15 update operators. Then,

$$\mathcal{T}_{U[1..0]} = \left\{ \begin{array}{c} \varepsilon \\ \varepsilon \\ \varepsilon \end{array} \right\}, \ \ldots, \ \mathcal{T}_{U[1..14]} = \left\{ \begin{array}{l} \overset{1}{\mathtt{a}}\ \overset{5}{\mathtt{a}}\ \overset{9}{\mathtt{a}}\ \overset{10}{\mathtt{b}}\ \overset{13}{\mathtt{c}} \\ \overset{2}{\mathtt{b}}\ \overset{3}{\mathtt{a}}\ \overset{8}{\mathtt{b}} \\ \overset{4}{\mathtt{a}}\ \overset{6}{\mathtt{c}}\ \overset{7}{\mathtt{b}}\ \overset{11}{\mathtt{c}}\ \overset{12}{\mathtt{b}}\ \overset{14}{\mathtt{b}} \end{array} \right\}, \ \mathcal{T}_{U[1..15]} = \left\{ \begin{array}{l} \overset{1}{\mathtt{a}}\ \overset{5}{\mathtt{a}}\ \overset{9}{\mathtt{a}}\ \overset{10}{\mathtt{b}}\ \overset{13}{\mathtt{c}} \\ \overset{2}{\mathtt{b}}\ \overset{3}{\mathtt{a}}\ \overset{8}{\mathtt{b}}\ \overset{15}{\mathtt{c}} \\ \overset{4}{\mathtt{a}}\ \overset{6}{\mathtt{c}}\ \overset{7}{\mathtt{b}}\ \overset{11}{\mathtt{c}}\ \overset{12}{\mathtt{b}}\ \overset{14}{\mathtt{b}} \end{array} \right\}$$

where the superscript i over each character a in the k-th text implies that $U[i] = (k, a)$. For instance, $U[15] = (2, \mathtt{c})$ and hence $\mathtt{c}$ was appended to the 2nd text $T_2 = \mathtt{bab}$ in $\mathcal{T}_{U[1..14]}$, yielding $T_2 = \mathtt{babc}$ in $\mathcal{T}_{U[1..15]}$.

If there is no restriction on U like the one in the example above, then U is called *fully-online*. If there is a restriction on U such that once a new character is appended to the k-th text, then no characters will be appended to its previous $k-1$ texts, then U is called *semi-online*. Hence, any semi-online sequence of update operators is of form $(1, T_1[1]), \ldots, (1, T_1[|T_1|]), \ldots, (K, T_K[1]), \ldots, (K, T_K[|T_K|])$.

Section 4.3 reviews previous algorithms which incrementally construct the DAWG and the suffix tree for a growing text collection in the semi-online setting. Section 4.4 proposes our new algorithm which incrementally construct the suffix tree for a text collection in the fully-online setting, respectively.

4.3 Fully-online version of DAWG and Weiner's suffix tree algorithm

Blumer et. al. [14, 15] and Crochemore [22] introduced the DAWG, also called suffix automaton, and gave a DAWG construction algorithm for a collection of semi-online texts. Their DAWG construction algorithm is very closely related to Weiner's reverse right-to-left suffix tree construction algorithm [25, 42, 50, 68]. In fact, both algorithm build dual structures and each exposes different parts of these structures, where the collection of semi-online left-to-right text inputs to the DAWG algorithm can be perceived as the same texts reversed right-to-left inputs to Weiner's suffix tree algorithm. Blumer et al.'s algorithm does not require a terminating \$ symbol and it was noted that the set of nodes of the DAWG and the reverse string's suffix tree coincide if the terminator symbols are present in both sets of inputs.

4.3.1 Semi-online construction of Weiner's suffix trees and DAWGs

We briefly explain how the suffix tree of a collection of semi-online right-to-left texts can be built by using Weiner's algorithm. For convenience, we assume that there is an auxiliary node $\perp$ that is the parent of the root r. We also assume that the edge from $\perp$ to r is labeled with *any* character c from Σ, $\mathcal{W}_c(\perp) = r$, and $slink(r) = \perp$. Assume that we have constructed $STree(\{T_1\$_1, \ldots, T_{K-1}\$_{K-1}\})$ in which all the hard W-links have been constructed and the Boolean indicator w have been appropriately maintained. Now we process the k-th and extend it from right-to-left. Since the end-marker $\$_k$ is a unique character, a new leaf representing $\$_k$ is created. Suppose we have inserted the leaves for the suffixes of $T_k\$_k$ with $T_k \in \Sigma^*$. The leaf that represents the k-th text $T_k\$_k$ is called the *handle leaf* for $T_k\$_k$. Now we are to prepend a new character a and insert the extended text $aT_k\$_k$ to the tree. We begin with the handle leaf ℓ

for $T_k\$_k$. We walk up from the handle leaf ℓ until finding the lowest explicit ancestor u' of ℓ which has hard W-link $\mathcal{W}_a(u')$ defined for the added character a. Also, let u be the lowest explicit ancestor of ℓ such that $\mathsf{w}_a u = 1$. Note that u is a descendant of u'. Let b be the first character of the path label from u' to ℓ. We move to the node $v' = au'$ using the hard W-link $\mathcal{W}_a(u')$, and let $v'' = au'by$ be the child of v' below the edge whose label begins with b, where $y \in \Sigma^*$. There are two cases: (1) If $|v''| - |v'| = |au'by| - |au'| = |by| > |u| - |u'|$, then we create a new explicit node $v = v''[1..|u| + 1]$ and set $\mathcal{W}_a(u) = v$. (2) Otherwise ($|by| = |u| - |u'|$), then there already exists an explicit node $v''[1..|u| + 1]$ and let v be this node. In both cases, we insert a new leaf ℓ' representing $aT_k\$_k$ as a child of v, and create a new hard W-link $\mathcal{W}_a(\ell) = \ell'$. This insertion point v for ℓ' represents the longest prefix of $aT_k\$_k$ that appears at least twice in the updated text collection, and hence, v is sometimes called as the *longest repeating prefix* of $aT_k\$_k$. Let s be any node in the path from u to ℓ such that $s \neq u$ (if any). In the suffix tree before the text $T_k\$_k$ was extended with a, we had $\mathsf{w}_a(s) = 0$. Now in the updated suffix tree, we update $\mathsf{w}_a(s) = 1$ due to the insertion of the new handle leaf ℓ' which represents $aT_k\$_k$. Also, node s gets a new soft W-link $\mathcal{W}_a(s) = \ell'$. These updates are common to both of Cases (1) and (2). There can be further updates in Case (1): Let s' be any node in the path from u' to u such that $s' \neq u'$ and $s' \neq u$ (if any). In the suffix tree before the text $T_k\$_k$ was extended with a, node s' had a soft W-link $\mathcal{W}_a(s') = v''$. Now in the updated suffix tree, this soft W-link is redirected as $\mathcal{W}_a(s') = v$. Also, the soft W-link $\mathcal{W}_a(u) = v''$ in the previous suffix tree gets redirected and becomes the hard W-link $\mathcal{W}_a(u) = v$ in the updated suffix tree. See Figure 4.2 for illustration.

Weiner's original algorithm is designed for a single right-to-left text, and for each prepended character a to the text $T\$$, the number of internal explicit nodes from the leaf for T to its lowest ancestor u' for which hard W-link $\mathcal{W}_a(u')$ exists can be amortized constant. This amortization argument is based on the fact that the depth of the path

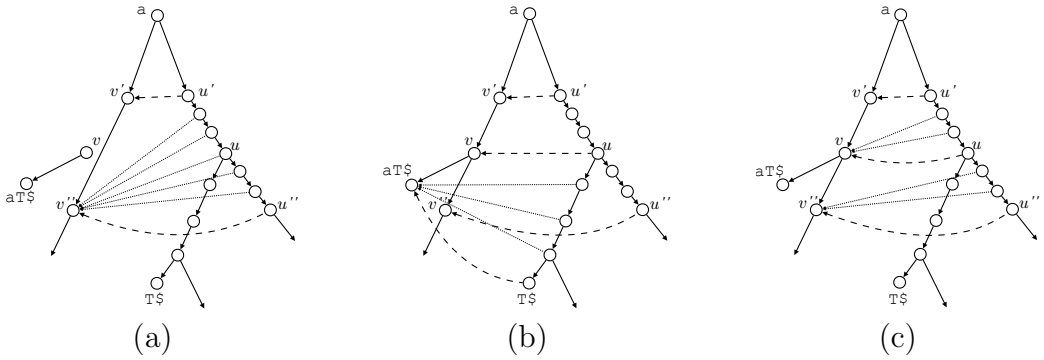

Figure 4.2: Extending the text $T_k\$_k$ to $aT_k\$_k$. Soft W-links are shown short-dashed and hard W-links are shown long-dashed. (a) relevant existing W-links before extending. (b) New W-links pointing to $aT_k\$_k$ are created from all nodes on the path between $T_k\$_k$ up to u. (c) Existing W-links pointing to v'' from all nodes on the path between u up to u' are redirected to point to v instead of v''. The new hard W-link $\mathcal{W}_a(T_k\$_k) = aT_k\$_k$ and redirected hard W-link $\mathcal{W}_a(u) = v$ have corresponding nodes on the path to $aT_k\$_k$, while all the other new soft W-links involved point to $aT_k\$_k$ and the redirected soft W-links involved point to v. The new node v also adopts all the outgoing W-links from v'' (not shown).

from the root to the handle leaf ℓ' representing the extended text $aT\$$ is by at most one larger than that of the path from the root to the handle leaf ℓ representing $T\$$. This property holds also in the semi-online setting, since while the kth text $T_k\$_k$ is being extended from right to left, other texts remain static and thus do not change the topology of the suffix tree. Hence, we can build $STree(\mathcal{T})$ for a collection of semi-online left-to-right texts in $O(n \log \sigma)$ time and $O(n)$ space.

Blumer et al. [14] showed how the DAWG for a collection $\mathcal{S}$ of semi-online left-to-right texts can be built in $O(n \log \sigma)$ time. Recall that each DAWG node represents an equivalence class of substrings which have the same ending positions in the texts. Appending a new character a to the currently processed text $\$_k S_k$ can affect some equivalence class under the current text collection. This can cause splitting an existing node into two nodes. Let w be the node that gets split and w' be the copy of this node w. The original node w will contains longer substrings than the copy w'. The longest element belonging to w' is the *longest repeating suffix* X of $\$_k S_k a$ in the updated text collection, and any element of w that is shorter than X will belong to w'. Eventually,

any element of w that is longer than X remains in w. This node split operation can be done by redirecting corresponding in-coming edges from w to w'. The key argument in the time analysis of Blumer et al.'s algorithm is that this cost of redirecting in-coming edges can also be amortized constant per added character a. Observe that this update is exactly the same as the above-mentioned update of the suffix tree for the corresponding right-to-left text collection. For instance, the longest repeating suffix of $\$_k S_k a$ for the current left-to-right text collection is the reverse of the longest repeating prefix of $aT_k\$_k$ for the corresponding right-to-left text collection. Also, redirecting those in-coming edges in the DAWG are exactly the same as updating a soft W-link to a hard one and redirecting soft W-links, in the suffix tree of the corresponding right-to-left texts (recall Case (1) above). Consequently, we can build the DAWG for a collection of semi-online left-to-right texts in $O(n \log \sigma)$ time and $O(n)$ space as well.

4.3.2 Fully-online construction of Weiner's suffix trees and DAWGs

In this subsection, we consider how to maintain the suffix tree for a collection of K texts which grow from right to left in a fully-online manner. This means that we will have to maintain K handle leaves for the K texts simultaneously. We also consider how to maintain the DAWG for a collection of K texts which grow from left to right in a fully-online manner.

Unfortunately, the identical amortization argument in both algorithms does not carry over in the fully-online setting. However, we will show next that Weiner's algorithm can be modified to work within the desired $O(n \log \sigma)$ time and $O(n)$ space bounds with the aid of σ nearest marked ancestor (NMA) data structures of total size $O(n)$, where σ denotes the number of all distinct characters appearing in the texts in the collection. Moreover, the same data structures can provide access to the DAWG edges, which cannot be maintained explicitly within our bounds, in $O(\log \sigma)$ time per

edge query.

We will use the following NMA data structure as a building block of our algorithm.

Lemma 8 ([69]) *There exists an NMA data structure for a growing rooted tree, which supports the following operations in amortized $O(1)$ time each: 1) find the NMA of a given node; 2) insert an unmarked node; 3) mark an unmarked node. This NMA data structure requires linear space in the size of the tree.*

Suppose that we have $STree(\mathcal{T}_{U[1..i-1]})$ for a fully-online right-to-left text collection $\mathcal{T}_{U[1..i-1]}$ and assume $U[i] = (k, a)$, i.e., the kth text $T_k\$_k$ gets extended with a new character a being prepended to it. As in the case with the semi-online texts, some new soft and hard W-links are created in the updated $STree(\mathcal{T}_{U[1..i]})$. Fortunately, the number of such newly created W-links are bounded by the size of the resulting suffix tree, which is $O(n)$. However, the number of *redirected* soft W-links, which are the same as the number of DAWG edges to be redirected, can be too numerous to be done within our desired bounds as the next lemma shows.

Lemma 9 *Weiner's suffix tree algorithm takes $\Theta(n \min(K, \sqrt{n}))$ time in the fully-online setting, where n is the total length of the K texts. Hence, for $K = \Theta(\sqrt{n})$ it also takes $\Theta(n\sqrt{n})$ time to explicitly maintain the soft W-links (equivalently, the DAWG secondary edges) in the fully-online setting. The lower bound holds for a constant alphabet.*

Proof 10 *To show that these bounds hold for constant alphabets, we here assume that each text in the collection terminates with the same end-marker \$. However, in our collection of texts each text will be distinct, so that each $T_k\$$ will be represented by a unique handle leaf.*

First, we consider a lower bound. Consider the following K right-to-left texts $\mathcal{T} = \{T_k = a^k\$ \mid 1 \leq k \leq K\}$ where $a \in \Sigma$ and each text terminates with a common end-

marker \$. *Suppose we have constructed the suffix tree of* $\mathcal{T}$ *in any order. Then, we prepend a new character* $c \in \Sigma$, *such that* $c \neq a$, *to each text* $T_k = a^k\$$ *in decreasing order of their length,* $k = K, \ldots, 1$. *Since we process each text in decreasing order of* k, *there are* $\Omega(k)$ *explicit nodes in the path from the handle leaf for* $T_k = a^k\$$ *to its lowest ancestor* $r = \varepsilon$ *(the root) for which hard W-link* $\mathcal{W}_c(r)$ *is defined. Hence, it takes* $\Omega(k)$ *time to naïvely walk up this path. Also, with the exception of the first longest text* T_K *that introduces* $\Omega(k)$ *new soft W-links, for all other* $k < K$, *there are* $\Omega(k)$ *soft W-links to be redirected along the way. Thus, there are* $\Omega(K^2)$ *edge re-directions in total, for all* k*'s. We then repeat the above procedure several times. At each repetition* i $(i > 1)$, *for each* k *in decreasing order it again takes* $\Omega(k)$ *time to walk up from the handle leaf for* $c^{i-1}a^k\$$ *until reaching its lowest ancestor* r *for which hard W-link* $\mathcal{W}_c(r)$ *is defined. Also, there are* $\Omega(k)$ *soft W-links to be redirected along the way. Thus, at each repetition* i, *it takes a total of* $\Omega(K^2)$ *time for all* k*'s, too,*

Let n *be the total length of the texts in the collection after performing the above procedure several times. The initial total length of the text collection* $\mathcal{T} = \{a^k\$ \mid 1 \leq k \leq K\}$ *is* $\frac{K(K+3)}{2}$. *We then append* c*'s to each of the* K *texts, and the text collection of total length finally becomes* n. *Hence, the number of iterations is* $(n - \frac{K(K+3)}{2})/K = \Theta(n/K - K)$, *which is* $\Theta(n/K)$ *in the case where* $K < \alpha\sqrt{n}$ *with some constant* α. *Since each iteration requires re-directions of* $\Omega(K^2)$ *soft W-links, it takes a total of* $\Omega(nK)$ *time in this case. Now consider the case where* $K > \alpha\sqrt{n}$. *In this case, we can apply the same procedure as above only to* $\alpha\sqrt{n}$ *texts in the collection, and the other* $K - \alpha\sqrt{n}$ *texts remain empty. This leads to* $\Omega(n\sqrt{n})$ *total work for re-directing soft W-links. Combining these two, we obtain an* $\Omega(n \min(K, \sqrt{n}))$ *lower bound.*

To see that this lower bound actually gives rise to the worse case in Weiner's algorithm, we can focus only on the time required for soft W-link re-redirection, since new edge insertions and node insertions are always accounted globally to be the total size of the the suffix tree, which is $O(n)$.

Recall that the number of soft W-link re-directions when appending a symbol a to text $T_k\$$ is no larger than the suffix tree depth of the handle leaf representing $T_k\$$, which is in turn smaller than the length of $T_k\$$. Also, the depth of the new leaf $aT_k\$$ is at most one more than the depth of leaf $T_k\$$ minus the number of edge re-directions that reduce depth of the current handle leaf associated with each of the K text, while the depth of all current handle leaves $T_i\$$, $i \neq k$, may also increase by at most one while updating $T_k\$$, by the insertion of the internal node off which the leaf $T_k\$$ is hanging above the handle leaf of T_i. Thus, each of the $O(n)$ symbols may increase by at most one the depth of all the K handle leaves. This depth increase was not an issue in the semi-online setting since previous $T_k\$$ are no longer updated and their handle leaves were no longer used. In the fully-online setting, this depth increase is problematic. The depth reduction argument gives an obvious $O(n)$ upper bound on the soft W-link re-directions while updating each of the K texts, which adds up to $O(Kn)$ overall upper bound.

The analysis will separate those short texts $T_k\$$, such that $|T_k\$_k| \leq \sqrt{n}$ from the longer texts. For the short texts, each time a symbol is prepended to a text $T_k\$$, the number of soft W-link edge re-directions is bounded by the length of each short text, which is at most $\sqrt{n}$, totaling at most $O(n\sqrt{n})$ such re-directions. For the long texts, we observe that there are at most $O(\sqrt{n})$ such long texts, and for each specific text, the total number of soft W-link edge re-directions is at most $O(n)$, totaling at most $O(Kn) \subseteq O(n\sqrt{n})$. Combining these bounds, we get the desired $\Theta(n \min(K, \sqrt{n}))$ tight bound. ∎

Remark 2 *To show that the bounds hold for a constant alphabet, we used the same end-marker \$ for all the texts in the proof of Lemma 9. We remark that the same arguments hold for the case where each text T_k is terminated with a unique end-marker $\$_k$, as we assume elsewhere in this chapter, since also in this case each text $T_k\$_k$ is represented by a unique handle text. We then use $K + 2$ characters in the lower bound*

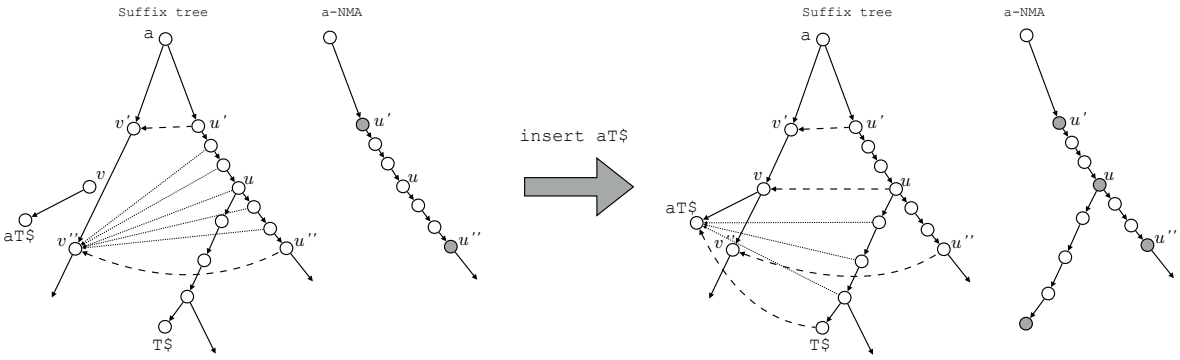

Figure 4.3: Extending the text $T_k\$_k$ to $aT_k\$_k$. Soft W-links are shown short-dashed and hard W-links are shown long-dashed. Gray nodes of NMA data structures mean marked nodes. Nodes with new hard W-link (of added character a) in the suffix tree are added to the NMA as marked nodes. Nodes with new soft W-link (of a) are added to the NMA as normal nodes.

example (the alphabet is $\{a, b, \$_1, \ldots, \$_K\}$*).*

To avoid the above-stated super-linear cost in Lemma 9, we shall only maintain hard W-links and will not explicitly maintain soft W-links. Instead of soft W-links we will maintain only the Boolean indicator $\mathsf{w}_a(v)$ that tells us whether a (soft or hard) W-link $\mathcal{W}_a(v)$ is defined or not. Once $\mathsf{w}_a(v)$ is set to 1, it remains 1 and does not need to be updated even when the corresponding soft W-link would have to be redirected.

Like in the semi-online setting, we here also go up from the leaf ℓ representing $T_k\$_k$ to its lowest ancestor u' for which $\mathcal{W}_a(u')$ is defined. The cost for walking up to the lowest ancestor u of ℓ for which $\mathsf{w}_a(u) = 1$ can be charged to the cost for creating new soft W-links (or equivalently, that for creating new corresponding DAWG edges), which is amortized constant per added character a. One problem remains: We would like to skip all explicit nodes s' in the path from node u to u', since naïvely walking up this path can be as costly as redirecting W-links $\mathcal{W}_a(s')$ for all such nodes s'. In so doing, we shall also maintain for each character σ an NMA data structure of Lemma 8 on the subtree of the suffix tree which consists of the two following disjoint sets of nodes: (1) the set of unmarked nodes v such that $\mathsf{w}_a(v) = 1$ and $\mathcal{W}_a(v)$ is a soft W-link, and (2)

the set of marked nodes v such that $\mathcal{W}_a(v)$ is a hard W-link. Our version of Weiner's algorithm will naïvely walk up the suffix tree from the leaf ℓ representing $T_k\$_k$ until the lowest node u such that $\mathsf{w}_a(u) = 1$, and from there it will jump to u' using the NMA data structure for the prepended character a. In what follows, we will denote this as the a-NMA data structure.

Theorem 8 *Given a fully-online sequence U of n update operators for a collection of K right-to-left texts $\mathcal{T}$, our version of Weiner's algorithm can update the suffix tree in a total of $O(n \log \sigma)$ time and $O(n)$ space.*

Proof 11 *The correctness of our algorithm should be clear from the above discussion.*

Let us analyze the time complexity. The algorithm will now still climb up the suffix tree from the currently focused leaf ℓ up to its lowest ancestor u with $\mathsf{w}_a(u) = 1$. From there, it would jump to its nearest ancestor u' of u having hard W-link $\mathcal{W}_a(u')$ defined in constant amortized time using an NMA query on the a-NMA data structure. Now we update the a-NMA data structure. If the insertion point v for the new leaf ℓ' representing $T_k\$_k$ is newly created (see Case (1) in the previous sub-section), then the soft W-link $\mathcal{W}_a(u)$ becomes hard. Hence, we mark node u in the a-NMA data structure. Otherwise, the W-link $\mathcal{W}_a(u)$ is already hard and hence u is already marked in the a-NMA data structure. Recall that each node s between the leaf ℓ and u obtain new soft W-links and hence $\mathsf{w}_a(s)$ is now set to 1. Hence, we insert an unmarked node for each s in the a-NMA data structure. Since the NMA data structure allows us to insert a new leaf in amortized constant time, we insert these unmarked nodes in increasing order of depth, from the child of u to the parent of ℓ contained in the path. We also spend $O(\log \sigma)$ time at each visited node for searching the appropriate NMA data structure. Overall, it takes a total of $O(n \log \sigma)$ time to construct the suffix tree for fully-online

Let us now analyze the space complexity. For each character $c \in \Sigma$, each marked node u in the c-NMA data structure corresponds to a unique hard W-link $\mathcal{W}_c(u)$. Also,

each unmarked node s in the c-NMA data structure corresponds to a unique soft W-link $\mathcal{W}_c(s)$. Since the total number of hard and soft W-links for all characters $c \in \Sigma$ is $O(n)$, the total size of the c-NMA data structures for all characters $c \in \Sigma$ is $O(n)$. ∎

Now we turn our attention to construction of the DAWG for a fully-online left-to-right text collection $\mathcal{S}$. Since our version of Weiner's algorithm does not explicitly maintain soft W-links, we do not have explicit representation of secondary edges of the DAWG for the left-to-right texts. However, the Weiner's suffix tree augmented with the NMA data structures indeed is implicit representation of the DAWG secondary edges:

Lemma 10 *Using Weiner's suffix tree augmented with the NMA data structures, we can simulate each soft W-link per query in amortized $O(\log \sigma)$ time.*

Proof 12 *A given node u has soft W-link $\mathcal{W}_a(u)$ for a given character a iff $\mathsf{w}_a u = 1$ and $\mathcal{W}_a(u)$ is not a hard W-link. Suppose u has soft W-link $\mathcal{W}_a(u)$. We query the NMA u' of u in the a-NMA data structure. Let b be the first character of the path label from u' to u. We follow the hard W-link $\mathcal{W}_a(u') = v'$, and find the out-going edge of v' whose edge label begins with b. Then, the child v'' of v' below this edge is the destination of the soft W-link $\mathcal{W}_a(u)$. The time for the NMA query is amortized to $O(1)$ and finding the appropriate a-NMA data structure and the appropriate out-going edge of v' takes $O(\log \sigma)$ time each.* ∎

The next corollary immediately follows from Theorem 8 and Lemma 10.

Corollary 9 *Given a fully-online sequence U of n update operators for a collection of K left-to-right texts, the DAWG can be maintained in a total of $O(n \log \sigma)$ time and $O(n)$ space with $O(\log \sigma)$ query time for an out-going DAWG edge.*

4.4 Fully-online version of Ukkonen's suffix tree algorithm

Ukkonen's algorithm [65] constructs the suffix tree of a given text in an online manner, from left to right. In this section, we show how Ukkonen's algorithm can be extended to maintain the suffix tree for a fully-online left-to-right text collection. We will do so by first explaining that Ukkonen's algorithm can readily be extended to the semi-online setting. Then, we will describe some difficulties in extending Ukkonen's algorithm to our fully-online setting, and finally we will present how to overcome these difficulties achieving $O(n \log \sigma)$-time algorithm.

4.4.1 Semi-online left-to-right suffix tree construction

Ukkonen's algorithm [65] can easily be extended to incrementally construct the suffix tree for multiple texts in the semi-online setting.

Let U be a semi-online sequence of n update operators such that the last update operator for each k ($1 \leq k \leq K$) is $(k, \#_k)$, where $\#_k$ is a special end-marker for the kth text in the collection. Also, assume that we have already constructed $STree(\mathcal{S}_{U[1..i-1]})$ and that the next update operator is $U[i] = (k, a)$. Thus a new character a is appended to the text S_k and it becomes $S_k a$.

In updating $STree(\mathcal{S}_{U[1..i-1]})$ to $STree(\mathcal{S}_{U[1..i]})$, we have to assure that all suffixes of the extended text $S_k a$ will be represented by $STree(\mathcal{S}_{U[1..i]})$. These suffixes are categorized to three different types:

Type-1 The suffixes of $S_k a$ that are longer than $lrs_{\mathcal{S}_{U[1..i-1]}}(S_k)a$.

Type-2 The suffixes of $S_k a$ that are not longer than $lrs_{\mathcal{S}_{U[1..i-1]}}(S_k)a$ and are longer than $lrs_{\mathcal{S}_U[1..i]}(S_k a)$.

Type-3 The suffixes of $S_k a$ that are not longer than $lrs_{\mathcal{S}_{U[1..i]}}(S_k a)$.

The suffixes of $S_k a$ are inserted in decreasing order of length.

The Type-1 suffixes are maintained as follows. Let s be any suffix of S_k which is represented by a leaf of $STree(\mathcal{S}_{U[1..i-1]})$. Since s is a non-repeating suffix of S_k in $\mathcal{S}_{U[1..i-1]}$, sa is a non-repeating suffix of $S_k a$ in $\mathcal{S}_{U[1..i]}$, which implies that sa will also be a leaf of $STree(\mathcal{S}_{U[1..i]})$. Based on this observation, the label of the in-coming edge of the leaf is represented by a pair $\langle k, b\rangle$ called an *open edge*, where b is the beginning position of the label of the in-coming edge in the kth text. We can retrieve the ending position of the edge label in constant time by looking at the current length of the kth text. This way, every existing leaf will then be "automatically" extended.

Hence, updating $STree(\mathcal{S}_{U[1..i-1]})$ to $STree(\mathcal{S}_{U[1..i]})$ reduces to inserting the Type-2 suffixes of $S_k a$ (note that the Type-3 suffixes of $S_k a$ already exists in the suffix tree). For this sake, the algorithm maintains an invariant which indicates the locus of $x = lrs_{\mathcal{S}_{U[1..i]}}(S_k)$ on $STree(\mathcal{S}_{U[1..i-1]})$ called the *active point*. Since x can be an implicit node, the algorithm maintains the canonical reference (v, c, ℓ) to x. For convenience, if x is an explicit node, then let its canonical reference be $(x, \varepsilon, 0)$. The update starts from the current active point x represented by its canonical reference pair, and the Type-2 suffixes of $S_k a$ are inserted in decreasing order of length, by using the chain of (virtual) suffix links. There are two cases:

I. If it is possible to go down from x with character a, then no updates to the tree topology are needed. The new active point is xa, and the reference to xa is made canonical if necessary. The update ends.

II. If it is impossible to go down from x with character a, then we create a new leaf. Let j be the beginning position of the suffix of $S_k a$ which corresponds to this new leaf. The following procedure is repeated until Case I happens.

   (a) If the active point x is on an explicit node, then a new leaf node s is created as a new child of x, with its incoming edge labeled by $\langle k, b\rangle$, where $b =$

$|S_k a| - |x| + 1$. The active point x is updated to $slink(x)$.

(b) If the active point x is on an implicit node, then x becomes explicit in this step. A new leaf node s is created as a new child of x with its incoming edge labeled by $\langle k, b\rangle$. Since the suffix link of the new explicit node x does not yet exist, we simulate the suffix link traversal as follows: Let (v_j, c_j, ℓ_j) be the canonical reference to x. First, we follow the suffix link $slink(v_j)$ of v_j, and then go down along the path of length ℓ_j from $slink(v_j)$ starting with character c_j. Let this locus be x'. Let v_{j+1} be the longest explicit node in this path. (i) If $|v_{j+1}| = |x'|$, then we firstly create the new suffix link $slink(x) = v_{j+1}$ for the new explicit node x. The active point x is updated to x' and is represented by canonical reference $(v_{j+1}, \varepsilon, 0)$. (ii) If $|v_{j+1}| < |x'|$, then the next active point is implicit. The active point x is updated to x' and is represented by canonical reference $(v_{j+1}, c_{j+1}, \ell_{j+1})$. The suffix link of x will be set to x' when x' becomes explicit in the next step.

The most expensive case is II-b-(ii). Since the path from v_{j+1} to x' contains at most $\ell_j - \ell_{j+1}$ explicit nodes, it takes $O((\ell_j - \ell_{j+1} + 1)\log\sigma)$ time to locate the next active point x' (note $\ell_j - \ell_{j+1} \geq 0$ holds). All the other operations take $O(\log\sigma)$ time. Hence, the total cost to insert all leaves (suffixes) for the kth text is $O(\sum_{j=1}^{n_k}(\ell_j - \ell_{j+1} + 1)\log\sigma) = O(n_k \log\sigma)$, where n_k is the final length of the kth text. Thus the amortized time cost for each leaf (suffix) for the kth text is $O(\log\sigma)$. Overall, it takes a total of $O(n\log\sigma)$ time to construct $STree(\mathcal{S}_U)$ for a semi-online sequence U of update operators. The space requirement is $O(n)$.

4.4.2 Difficulties in fully-online left-to-right suffix tree construction

The following observations suggest that it does not seem easy to extend Ukkonen's algorithm to our left-to-right fully-online setting:

A. **[Keeping track of active points]** Let $U[i] = (k, a)$ which updates the current kth text S_k to $S_k a$, and assume that we have just constructed $STree(\mathcal{S}_{U[1..i]})$. Recall that we defined the initial locus of the active point for $S_k a$ on $STree(\mathcal{S}_{U[1..i]})$ to be the longest repeating suffix of $T_k a$ in $\mathcal{S}_{U[1..i]}$. However, since U is fully-online, any other text T_h ($h \neq k$) in the collection would be updated by following update operators $U[r]$ with $r > i$. Then, the longest repeating suffix of $S_k a$ in $\mathcal{S}_{U[1..r]}$ can be much longer than that of $S_k a$ in $\mathcal{S}_{U[1..i]}$. In other words, some Type-1 suffixes of $S_k a$ in $\mathcal{S}_{U[1..i]}$ can become of Type-2 in $\mathcal{S}_{U[1..r]}$. What is worse, updating S_h can affect the longest repeating suffix of any other text in the collection as well.

B. **[Canonization of active points]** Even if we somehow manage to efficiently maintain the active point for each text in the collection, there remains another difficulty. Let j be the beginning position of the longest repeating suffix of $S_k a$ in $\mathcal{S}_{U[1..i]}$, and let (v_j, c_j, ℓ_j) be the canonical reference to this suffix. Let $U[i'] = (k, a')$ be the first update operator in U which updates the kth text after $U[i] = (k, a)$. Let (v'_j, c'_j, ℓ'_j) be the canonical reference to the longest repeating suffix of $S_k a$ in $\mathcal{S}_{U[1..i']}$, which is the "real" initial active point where insertion of the Type-2 suffixes should start at this i'th step. By the property of suffix trees $\ell'_j \geq \ell_j$ holds, and what is worse, this length ℓ'_j is unbounded by the number of Type-2 suffixes inserted at this i'th step. Thus, it is not clear whether the amortization technique we used for the semi-online construction works in our fully-online setting.

C. **[Maintaining leaf ownerships]** The phenomenon mentioned in Difficulty A

also causes a problem of how to represent the labels of the in-coming edges to the leaves. Assume that we created a new leaf w.r.t. an update operator (k, a), and let $\langle k, b_k \rangle$ be the pair representing the label of the in-coming edge to the leaf, where b_k is the beginning position of the edge label in the kth text. We say that the kth text S_k is the *owner* of the leaf. It corresponds to a Type-1 suffix of the kth text, but the leaf can later be extended by another growing text S_h. Namely, S_h can overtake the ownership of the leaf from S_k. After this happens, then the pair $\langle k, b_k \rangle$ has to be updated to $\langle h, b_h \rangle$, where b_h is the beginning position of the edge label in the hth text. Notice that this update may happen repeatedly.

4.4.3 Fully-online left-to-right suffix tree algorithms

Let us now consider how to construct the suffix tree for a fully-online left-to-right text collection. Our fully-online version of Ukkonen's algorithm works with the aid of the fully-online version of Weiner's algorithm proposed in Section 4.3. Namely, for a fully-online left-to-right text collection $\mathcal{S}$ with K texts, we build $STree(\mathcal{S})$ in tandem with $STree(\mathcal{T})$, where $\mathcal{T}$ is the set of reversed texts from $\mathcal{S}$ (i.e., $\mathcal{T} = \overline{\mathcal{S}}$). Since we use the fully-online version of Weiner's algorithm, as in Section 4.3, we assume that each text in $\mathcal{T}$ terminates with a special symbol $\$_k$, namely, $\mathcal{T} = \{T_1\$_1, \ldots, T_K\$_K\}$. This in turn implies that each text in $\mathcal{S}$ begins with $\$_k$, namely, $\mathcal{S} = \{\$_1 S_1, \ldots, \$_K S_K\}$, where $S_i = \overline{T_i}$ for $1 \leq i \leq K$.

In what follows, we will propose two alternative approaches. Suppose we have constructed $STree(S_{U[1..i-1]})$. Given the ith update operator $U[i] = (k, a)$, the first one called the *forward approach* traverses a chain of (virtual) suffix links in a forward manner and inserts new leaves of the updated text $\$_k S_k a$ in decreasing order of the lengths of the suffixes of $\$_k S_k a$. This forward approach is a direct extension of Ukkonen's original algorithm. The second one called the *backward approach* traverses a chain of (virtual) suffix links in a backward manner and inserts new leaves in increasing order of

the lengths of the suffixes of $\$_k S_k a$. This backward approach can be seen an extension of Breslauer and Italiano's algorithm [16] which was originally proposed for real-time suffix tree construction for a single left-to-right text.

Forward approach

In this subsection, we present our forward approach to update $STree(S_{U[1..i-1]})$ to $STree(S_{U[1..i]})$. The key notions in this forward approach are *swapping active points* and *tight connections between active points and leaf ownerships*. In what follows we will explain these notions in full details.

Let us first consider maintaining active points (Point A). This is indeed closely related to maintaining leaf ownerships (Point C). We will for now put it aside the cost for maintaining leaf ownerships, and will focus on describing how active points can affect ownerships of leaves.

For a single right-to-left online text, the suffix links of the leaves form a single path from the longest leaf to the shortest one. On top of them we also consider a virtual suffix link from the shortest leaf to the active point.

We generalize the above notion to our fully-online text collection $\mathcal{S}$. Unlike the single text case, a leaf can represent a suffix of multiple texts in our fully-online setting. This implies that the suffix links of $STree(\mathcal{S})$ form a forest. Let $F_{\mathcal{S}}$ denote this forest. This forest is only conceptual, namely, in our algorithms to follow we will *not* explicitly maintain it. However, the forest gives us more insights into Points A and C. Formally, the forest $F_{\mathcal{S}}$ is a set of maximal trees such that each maximal tree SLT in $F_{\mathcal{S}}$ satisfies:

- the root of SLT is the locus (an implicit or an explicit internal node) of the active point of a text,
- the other nodes of SLT are leaves of $STree(\mathcal{S})$, and
- the (reversed) edges of SLT are suffix links of $STree(\mathcal{S})$ (if the root of SLT is an implicit node, then the (reversed) edges from the root to its children are virtual

suffix links from the children).

Since a leaf of $STree(\mathcal{S})$ can be a suffix of multiple texts, there are multiple choices for the owner of each leaf. Our choice of the owner of a leaf is either

(R1) the text that created the leaf, or

(R2) the last text whose active point has extended the leaf.

Regarding Rule (R2) above, we will soon describe in more details how the active point of a text can extend an existing leaf.

Suppose that we have constructed $STree(\mathcal{S}_{U[1..i-1]})$ and that we are given an update operator $U[i] = (k, a)$ which appends new character a to text $\$_k S_k$.

If the active point of $\$_k S_k$ is not on a leaf of $STree(\mathcal{S}_{U[1..i-1]})$, then the suffix tree is updated as in the semi-online setting and there are no changes on the ownerships of the leaves. Hence, in what follows we consider the case where the active point of $\$_k S_k$ is on a leaf of $STree(\mathcal{S}_{U[1..i-1]})$.

Let s be the leaf of $STree(\mathcal{S}_{U[1..i-1]})$ where the active point of the text $\$_k S_k$ lies. Let SLT denote the suffix link tree in $F_{S_{U[1..i-1]}}$ that contains this node s, and let P_i be the path from s to the root of SLT. Also, let $\mathcal{O}_i$ be the set of texts which are the owners of the suffix tree leaves in P_i. Finally, let L_i be the list of all nodes u in the path from the *parent* of s to the root of SLT such that the active point of some text in $\mathcal{O}_i$ lies on u. For each $1 \leq x \leq m = |L_i|$, let $u_x = L_i[x]$. For convenience, let $u_0 = s$. For each $1 \leq x \leq m$, let k_x denote the text id of the owner of u_x. Then, due to the way how the ownerships of leaves are defined by Rules (R1) and (R2) above, for every $1 \leq j < m$ the owner of every leaf between u_{j-1} and u_j is the k_jth text in the collection. See also the left diagram of Figure 4.4 for illustration.

Now we describe how the ownerships of leaves and the active points of texts can change when a new character is appended to a text in the fully-online setting. We begin with the first node $u_1 = s$ in the list L_i whose current owner is text $\$_{k_1} S_{k_1}$. See

also the left diagram of Figure 4.4. Since $\$_k S_k$ now gets extended to $\$_k S_k a$, the active point of this text *extends* the suffix tree leaf u_1. Then, the extended leaf u_1 no more represents a suffix of its original owner $\$_{k_1} S_{k_1}$. This implies that the new owner of this suffix tree leaf u_1 is $\$_k S_k a$. The same happens to all leaves in the path up to u_1. Then, we *swap* the active points of texts $\$_k S_k$ and $\$_{k_1} S_{k_1}$. We continue the same procedure recursively for the other nodes $u_2, \ldots, u_m$ in the list L_i, and finally the new owner of each leaf in the path P_i becomes the updated kth text $\$_k S_k a$. After reaching the root of SLT, we possibly create new edges labeled with a following virtual suffix links, and finally arrive at the new locus of the active point for the updated kth text $\$_k S_k a$. This operation may split the original suffix link tree SLT into some smaller suffix link trees (see also Figure 4.4).

Lemma 11 *The above procedure correctly maintains the active points of texts and the leaf ownerships under Rules (R1) and (R2).*

Proof 13 *It is clear that the above procedure correctly maintains the leaf ownerships under Rules (R1) and (R2).*

Let $\$_{k_j} S_{k_j}$ be any text in $\mathcal{O}_i$. After swapping the active points of $\$_k S_k a$ and those texts in $\mathcal{O}_i$, the locus of the active point of $\$_{k_j} S_{k_j}$ is one character above the suffix tree leaf (say u) that has just been extended by $\$_k S_k a$. By the definition of list L_i, this leaf u before extension *was the longest leaf whose previous owner was $\$_{k_j} S_{k_j}$. Hence, the string depth of the new active point of $\$_{k_j} S_{k_j}$ is at least $|u|-1$. Also, it cannot be larger than $|u|-1$, since otherwise it contradicts with the definitions of $\mathcal{O}_i$ and L_i (see also Figure 4.4). Hence, the above procedure of swapping active points correctly maintains the active points of the texts in the collection.* ∎

Now we wish to maintain leaf ownerships as described above. However, the next lemma shows that it requires super-linear cost to explicitly maintain leaf ownerships.

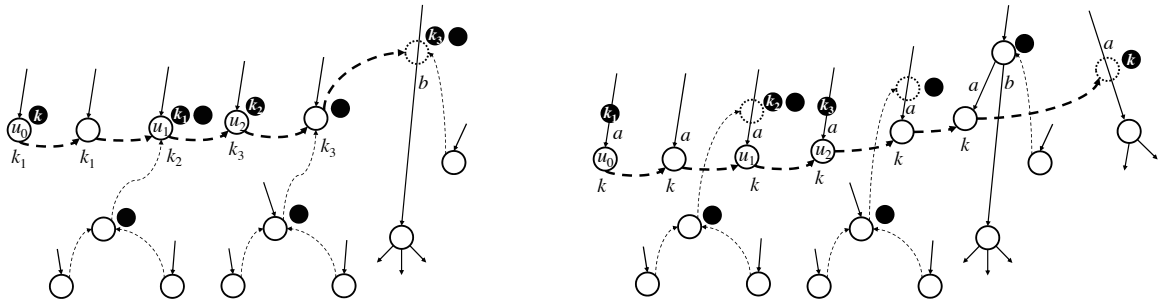

Figure 4.4: The left diagram depicts a suffix link tree in the forest before the ith update with update operator $U[i] = (k, a)$, where the solid and broken arrows respectively represent suffix tree edges and suffix links, and the white and black circles respectively represent suffix tree leaves and active points. The dotted circle represents the root of the suffix link tree (which is an implicit node in this case). The suffix link path P_i of interest is shown with bold broken arrows, where the staring node is $s = u_0$. The integers k_j below each leaf shows the current owner of the leaf, and hence $\mathcal{O}_i = \{\$_k S_k, \$_{k_1} S_{k_1}, \$_{k_2} S_{k_2}, \$_{k_3} S_{k_3}\}$. The integer k_j in each black circle implies that it is the active point for text $\$_{k_j} S_{k_j}$. The black circles without text id's are the active points of texts which are not in $\mathcal{O}_i$. The right diagram shows how it looks after the text $\$_k S_k$ has been extended to $\$_k S_k a$ with a new character a. Since its active point has extended the leaf u_0 with a, text $\$_k S_k$ becomes the new owner of every leaf in the path P_i. In the meantime, we swap the active point for text $\$_k S_k a$ with the active points of texts in $\mathcal{O}_i$, in the order they appear in the path P_i. After the active point of text $\$_k S_k a$ and that of the last text in the path (which in this figure is $\$_{k_3} S_{k_3}$) have been swapped, we possibly create new leaves (in this figure we create just one new leaf), and eventually we find the new locus for the active point for the updated text $\$_k S_k a$. Since all the leaves in the path P_i have been extended by the new character a, this path breaks away from the original suffix link tree. As a result, we obtain several smaller suffix link trees.

Lemma 12 *There is a left-to-right fully-online collection of K texts of total length n for which explicitly maintaining leaf ownerships requires $\Omega(\frac{n^2}{K})$ time.*

Proof 14 *Consider an initial text collection $\mathcal{S} = \{\$_1, \ldots, \$_K\}$. We will update this text collection in i rounds so that in each jth round the same character a_j is appended to each text. The order of the texts to which a_j is appended is arbitrary in each round. Thus, after the jth round, the text collection becomes of form $\{\$_1 a_1 \cdots a_j, \ldots, \$_K a_1 \cdots a_j\}$. We also assume that $a_j \neq a_h$ for any $1 \leq j \neq h \leq i$. This implies that in each jth*

round, we will have j leaves representing common suffixes $a_1 \cdots a_j$, $a_2 \cdots a_j$, ..., a_j.

Notice that during the jth round, the ownership of each such leaf has to be updated K times since each such leaf is shared by the K texts. Therefore, the total number of updates for the leaf ownership after the final ith round is at least

$$K(1 + 2 + \cdots + i) = \frac{Ki(i+1)}{2}. \tag{4.1}$$

Since n is the total length of the resulting text collection after the ith round, we get $n = K(i+1)$. Hence, $i = \Theta(\frac{n}{K})$. Plugging this into equation 4.1, we obtain the desired lower bound $\Omega(\frac{n^2}{K})$. ∎

The above $\Omega(\frac{n^2}{K})$ lower bound requires us a super-linear cost for explicit leaf ownership maintenance when $K = o(n)$. Indeed, $K = o(n)$ is the only meaningful case in our fully-online problem: If $K = \Theta(n)$, then each of the K texts is of constant size and hence a naïve algorithm would update the suffix tree in constant time per each text no matter how they are updated, resulting in an $O(n)$-time construction anyway. Hence, in what follows, we will only consider the case where $K = o(n)$.

Due to Lemma 12, we shall not explicitly maintain leaf ownerships in our fully-online algorithm. However, when swapping the active point of the kth text with those of the texts in the set $\mathcal{O}_i$, we need to know the owner of the leaf that has just been extended by the active point of the kth text. We also need to know the set $\mathcal{O}_i$ of texts which are the owners of the leaves in the path P_i, and need to know the list L_i of leaves where those active points currently lie. For this sake we use the aid of our version of Weiner's algorithm for fully-online right-to-left construction. Namely, we build $STree(\mathcal{S}_{U[1..i]})$ in tandem with $STree(\mathcal{T}_{U[1..i]})$ for each increasing $i = 1, \ldots, n$. For simplicity, we will call the left-to-right fully-online suffix tree $STree(\mathcal{S}_{U[1..i]})$ as the *Ukkonen tree* and the right-to-left fully-online suffix tree $STree(\mathcal{T}_{U[1..i]})$ as the *Weiner tree*.

Below we show key observations that connect our versions of Weiner's algorithm and Ukkonen's algorithm in the fully-online setting. For each node v of the Weiner tree, let $\mathsf{w\_deg}(v)$ denote the number of (soft or hard) W-links from v, namely, $\mathsf{w\_deg}(v) = |\{c \in \Sigma \mid \mathsf{w}_c(v) = 1\}|$.

Lemma 13 *Let u be any leaf in the list L_i of the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$. Then, there exists an explicit node v of the Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$ such that (1) $v = \overline{u}$, (2) v is in the path from the root to the leaf representing $T_k\$_k$, and (3) $\mathsf{w\_deg}(v) = 0$.*

Proof 15 *Since u is a leaf of the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$, it is a suffix of the text $\$_k S_k$ to which a new character a will be appended. Hence $v = \overline{u}$ is a prefix of the reversed text $T_k\$_k$, and is located on the path from the root to the leaf $T_k\$_k$ in the Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$. By the definition of the list L_i, the active point of some other text (say $\$_h S_h$, with $h \neq k$) lies on the leaf u in the Ukkonen tree, which implies that u is the longest suffix of $\$_h S_h$ that occurs at least twice in the left-to-right collection. Since each left-to-right text begins with a distinct $\$$ symbol, there must be at least two distinct characters that immediately precede occurrences of u. This in turn implies that there are at least two distinct characters that immediately follow occurrences of $v = \overline{u}$ in the right-to-left text collection, and hence $v = \overline{u}$ is an explicit node in the Weiner tree. To prove (3) assume on the contrary that $\mathsf{w\_deg}(v) > 0$, and let c be any character such that $\mathsf{w}_c(v) = 1$. Since $cv = c\overline{u}$ is a substring of some text in the right-to-left collection $\mathcal{T}_{U[1..i-1]}$, uc is a substring of some text in the left-to-right collection $\mathcal{S}_{U[1..i-1]}$. However, this contradicts that u is a leaf of the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$. Hence $\mathsf{w\_deg}(v) = 0$.* ∎

As was shown in Section 4.3, when we update the Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$ to $STree(\mathcal{T}_{U[1..i]})$ with update operator $U[i] = (k, a)$ which prepends character a to text $T_k\$_k$, we walk up from the leaf $T_k\$_k$ until finding the first node with a (soft or hard) W-link w.r.t. a defined. Since the total cost of walking up these paths for all characters

prepended to the right-to-left texts is linear in the final total length n of all texts, the number of nodes in the list L_i for $1 \leq i \leq n$ is also linear in n.

Notice that not every explicit node v with $\mathsf{w\_deg}(v) = 0$ in the path from the leaf $T_k\$_k$ to the root of the Weiner tree corresponds to a leaf in the list L_i on the Ukkonen tree. However, as was shown above, we can afford to check each such explicit node v in total linear time.

The next lemma shows how to maintain correspondence between these nodes in the Weiner tree and the Ukkonen tree.

Lemma 14 *We can maintain correspondence between each node v of the Weiner tree with $\mathsf{w\_deg}(v) = 0$ and its corresponding leaf u in the Ukkonen tree in $O(n \log \sigma)$ total time.*

Proof 16 *Let v be any node of the Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$ with $\mathsf{w\_deg}(v) = 0$. Suppose we have maintained correspondence between v and its corresponding leaf u in the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$. This correspondence is maintained by bidirectional links between the two trees.*

Now suppose we are given an update operator $U[i] = (k, a)$ which appends a new character a to $\$_k S_k$ and prepends a to $T_k\$_k$. There are three cases to consider.

(a) If the active point of the kth left-to-right text extends a leaf of the Ukkonen tree: In this case, as was described previously and was illustrated in Figure 4.4, the leaves in the path P_i get extended by the new character a that was appended to the kth left-to-right text $\$_k S_k$. This implies that v in the updated Weiner tree $STree(\mathcal{T}_{U[1..i]})$ does not correspond to a leaf in the updated Ukkonen tree $STree(\mathcal{S}_{U[1..i]})$. Thus, we remove the bidirectional link that connects v and the corresponding leaf in the Ukkonen tree.

(b) If the active point of the kth text catches up a leaf u of the Ukkonen tree: Since u is a leaf whose current owner is another text $\$_h S_h$ with $h \neq k$, u is a suffix

of at least two distinct left-to-right texts in the updated collection $\mathcal{S}_{U[1..i]}$. Hence, $\overline{u}$ is a prefix of at least two distinct right-to-left texts in the updated collection $\mathcal{T}_{U[1..i]}$, and hence is represented by an explicit node in the updated Weiner tree $STree(\mathcal{T}_{U[1..i]})$. Let v be this explicit node. Moreover, since u is the locus of the active point of $\$_k S_k a$, u is the longest repeating suffix of $\$_k S_k a$ and hence $v = \overline{u}$ is the longest repeating prefix of $aT_k\$_k$. This node v is exactly the insertion point of the new leaf $aT_k\$_k$ in the Weiner tree. Hence, we can find the locus of $v = \overline{u}$ during the updates of the Weiner tree and can easily create a bidirectional link between v and u.

(c) Otherwise, there are no changes in the correspondence and hence no maintenance of bidirectional links is needed.

In both cases (a) and (b), the costs can be charged to the construction of the Weiner tree which takes total $O(n \log \sigma)$ time. ∎

In Lemmas 13 and 14 we have shown how to efficiently find those suffix tree leaves in the list L_i of the Ukkonen tree with the aid of the Weiner tree. What remains is how to find each text in the set $\mathcal{O}_i$ of owners of the leaves in the list L_i. The next lemma shows yet another application of the Weiner tree for this purpose.

Lemma 15 *With the aid of the Weiner tree, we can find the owner of each leaf in the list L_i in total $O(n \log \sigma)$ time for all $1 \leq i \leq n$.*

Proof 17 *In each internal explicit node of the Weiner tree, we store the id of the text which created the oldest leaf in the subtree rooted at this internal explicit node. This can be easily maintained in $O(1)$ time per node: When we split an edge and create a new internal node, then we simply copy the text id stored in its unique child.*

Consider any update operator $U[i] = (k, a)$. Let u be any leaf in the list L_i of the Ukkonen tree and let v be its corresponding node in the Weiner tree (hence $v = \overline{u}$ and

it is an explicit node due to Lemma 13). Then, if the text id stored in v is h, then the hth text is the current owner of the leaf u in the Ukkonen tree. This is true in either case where the leaf u was created by the hth text and has never been extended by an active point, or the leaf u was last extended by the hth text. In both cases, the subtree rooted at $v = \overline{u}$ in the Weiner tree may contain leaves which correspond to suffixes of some other texts than the hth text, but in the Ukkonen tree the active points of these texts only caught up with the leaf u. Hence none of these texts is the one which created the leaf u, or the last one that has extended u. Therefore, the hth text is the current owner of u.

A careful consideration is required when the leaf u gets extended by the active point of text $\$_k S_k$. Now the extended leaf represents the extended string ua and its new owner is the kth text $\$_k S_k a$. As was shown in the proof for Lemma 14, in the Weiner tree the reversed extended string $a\overline{u}$ is represented by a new, different locus than the locus for $\overline{u}$. It is also possible that $a\overline{u}$ is on an implicit node in the Weiner tree at this stage, but it will become explicit when the active point of another text catches up the leaf ua in the Ukkonen tree. Thus, we will be able to return the text id k as the correct answer for a leaf ownership query when the active point of another text extends the leaf ua in future. ∎

In the above arguments we have shown that Difficulties A and C can be efficiently resolved by swapping active points and by neglecting explicit maintenance of leaf ownerships.

Meanwhile, this lazy maintenance of leaf ownership causes two more issues; Suppose that the active point of some text $\$_i S_i$ lies on an edge that leads to a leaf u, and that a new character a has been appended to this text. Let x be the string represented by the active point.

- The first question is how we can determine whether the active point can step forward along this edge by character a, or a new explicit node must be created

at the locus of x together with a new edge labeled with a. Since we do not know the owner of the leaf u, we are not able to answer the above question by a simple character comparison. However, this can be answered again by the aid of the Weiner tree. Recall that there is an explicit node representing the reversed string $\overline{x}$ in the Weiner tree and we know its locus through the updates of the Weiner tree. Now, the active point can step forward with character a if and only if the node $\overline{x}$ has a (soft or hard) W-link for character a. Hence, we can answer the above question in $O(\log \sigma)$ time. In case where we cannot step forward with character a, then we need to create a new edge leading to a new leaf. Instead of explicitly maintaining the owner of the leaf, we only maintain the first character a of this edge label. If the locus of the active point is on an edge, then we create a new explicit node u representing x in the Ukkonen tree. Now u has two out-going edges both leading to leaves, one of which is labeled with a as was described above. Since x was on an edge, there was a unique character, say b, such that $b \neq a$ and the W-link of node $\overline{x}$ for character b is defined in the Weiner tree. Thus the other out-going edge of u is labeled with b in the Ukkonen tree. Also, by storing the string depth in each active point, the whole label of the edge from the parent of u to u can be easily determined in constant time. Thus, we are able to eagerly maintain the whole label of every edge leading to an internal explicit node.

- The second question is how we can know that the active point catches up the leaf. In the preceding discussions, we only proved that we can find the owner of the leaf *after* we know that the active point has caught up the leaf. We observe that the active point catches up the leaf if and only if the Weiner tree node v representing $a\overline{x}$ is of Weiner degree zero, namely, the W-link of node v is undefined for any character. Hence, this question can also be answered by the aid of the Weiner tree in constant time.

The final issue in this forward approach is how to overcome Difficulty B on the cost for canonizing active points. The next lemma implies that the cost in our fully-online setting can indeed be amortized by a simple modification to the original amortization arguments in the semi-online setting.

Lemma 16 *The total cost for canonizing the active points for all K texts in a left-to-right collection $\mathcal{S}$ is $O(n \log \sigma)$.*

Proof 18 *Since we swap active points, the owner of each active point can change during the construction of the Ukkonen tree. However, our analysis below does not consider which text is the owner of each active point and hence it will lead us to simple arguments.*

Let A denote any active point and let (u_A, c_A, ℓ_A) denote the reference pair of A. We remark that in our fully-online setting, this reference pair may not be canonical, since some other text can split the out-going edge of node u_A whose label begins with c_A. The potential *of the active point A is ℓ_A of the string that hangs off from the explicit node u_A.*

Suppose we have constructed $STree(\mathcal{S}_{U[1..i-1]})$, and that we are given the ith update operator $U[i] = (k, a)$ which appends new character a to the kth text $\$_k S_k$. Also, suppose that A is the active point for $\$_k S_k$ at this stage. Now the algorithm finds the new locus for the active point A for the updated text $\$_k S_k a$, while possibly swapping several active points and inserting new leaves. In this event the algorithm traverses a chain of (virtual) suffix links. When a canonization is conducted after tracing a virtual suffix link, then the potential ℓ_A decreases at least one. Also, when the new locus of the active point A is found on the updated suffix tree $STree(\mathcal{S}_{U[1..i]})$, then the potential increases exactly by one with the new character a. Hence, the total number of canonizations performed for all n added characters is at most n.

Each canonization operation requires $O(\log \sigma)$ time to find the out-going edge whose

label begins with the corresponding character. Hence, the total cost for canonizations for all n characters is $O(n \log \sigma)$. ∎

Putting the above arguments all together, we have proven the following theorem.

Theorem 10 *Given a fully-online sequence U of n update operators for a collection of K left-to-right texts $\mathcal{S}$, our forward version of Ukkonen's algorithm can update the suffix tree in a total of $O(n \log \sigma)$ time and $O(n)$ space.*

Example 1 *Fig. 4.5 shows a snapshot of left-to-right fully-online suffix tree construction, where the $\$_i$ symbols are omitted for simplicity. Recall that we employ lazy maintenance of leaf ownership, and hence each character within a box is only imaginary and is not computed during the updates. Due to lazy representation of leaves, we do nothing to insert the Type-1 suffixes of S_1`b`. The active point of S_1 was on a leaf whose owner was S_2, and then it has extended the leaf. Hence, we swap the active points of S_1 and S_2. To start inserting the Type-2 suffixes in decreasing order of length, we first insert the longest Type-2 suffix* `abb` *at the locus of the active point of S_1. With the aid of the Weiner tree, we determine whether the active point can step forward along this edge by character* `b`*. In this case, the active point cannot step forward, and hence create a new internal node in the middle of this edge. After creating a new leaf from the new internal node and its in-coming edge with the first character label* `b`*, we determine the label of the in-coming edge of the new internal node using Weiner tree. Then the active point traces the virtual suffix link from the new internal node* `ab` *to node* `b`*. This virtual link can be computed by using the suffix link of node* `a`*. The next Type-2 suffix is* `bb`*, and the active point cannot step forward with* `b`*. Therefore we create a new internal node in the middle of this edge. Then the active point traces the virtual suffix link from the new internal node* `b` *to the root. The next shorter suffix* `b` *is Type-3, since we can step forward with character* `b` *from the root. Therefore, we move the active point from the root to node* `b` *that represents the longest repeating suffix of S_1`b`, and the reversed*

suffix link is set from root to the node `b`*. Since we have inserted all the Type-2 suffixes, the update finishes.*

Backward approach

In this subsection, we propose the backward approach which traces a chain of (virtual) suffix links in the reversed order and inserts new leaves in increasing order of their string lengths.

Suppose we have constructed $STree(\mathcal{S}_{U[1..i-1]})$ and we are now given an update operator $U[i] = (k, a)$. Consider the locus of the insertion point of the shortest Type-2 suffix of the updated text $\S_k S_k a$ in the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$. This locus corresponds to the suffix of $\$_k S_k a$ that is exactly one character longer than the longest Type-3 suffix $lrs_{S_{[U1..i-1]}}(\$_k S_k a)$ of $\$_k S_k a$ in the text collection $\mathcal{S}_{U[1..i-1]}$ before update. In the backward approach we first find this locus, and insert the Type-2 suffixes of the updated text $\$_k S_k a$ in increasing order of lengths. Since we trace the chain of suffix links backward, we use the reversed suffix links with character labels. In other words, we maintain *the hard W-links on the Ukkonen tree.*

We also remark that we do not need to swap active points in this backward approach, since we begin with the *shortest* Type-2 suffix. This somewhat simplifies the concept of the algorithm and might be an advantage over the forward counterpart presented in Section 4.4.3.

To find the canonical reference to the locus of the insertion point of the shortest Type-2 suffix of $\$_k S_k a$, we use the spanning tree of $DAWG(\mathcal{T}_{U[1..i]})$ which consists only of the primary edges. This tree consists of the longest paths from the source of the DAWG to its nodes, and hence, it coincides with the tree of *the reversed hard W-links of the Weiner tree* (this should not be confused with the hard W-links on the Ukkonen tree for backward suffix link traversals). For each $1 \leq i \leq n$, let $LPT(\mathcal{S}_{U[1..i]})$ denote this tree. By the property of DAWGs (and hence that of the equivalence relation), the following fact holds.

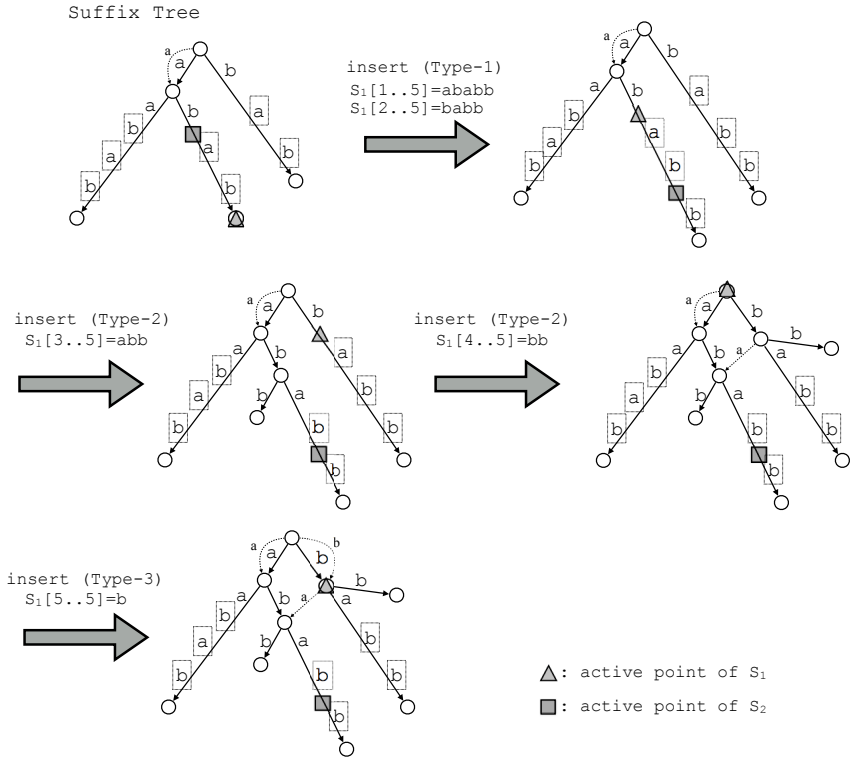

Figure 4.5: A snapshot of left-to-right fully-online suffix tree construction in the forward approach for Example 1. We update $STree(\mathcal{S})$ to $STree(\mathcal{S}')$ with $\mathcal{S} = \{S_1 = \texttt{abab}, S_2 = \texttt{aabab}\}$ and $\mathcal{S}' = \{S_1\texttt{b}, S_2\}$ (here the terminate symbols $\$_1$ and $\$_2$ are omitted for simplicity).

Fact 2 *For any $2 \leq i \leq n$, if an edge e is a primary edge of $DAWG(\mathcal{T}_{U[1..i-1]})$, then e is a primary edge of $DAWG(\mathcal{T}_{U[1..i]})$.*

We also use the following fact in our algorithm.

Fact 3 *For any substring x of texts in a left-to-right text collection $\mathcal{S}$, node x is branching (explicit) in $STree(\mathcal{S})$ iff node $[x]_{\mathcal{S}}$ is branching in $DAWG(\mathcal{S})$.*

Based on Fact 3, for each $1 \leq i \leq n$, we will maintain the NMA data structure $LPT(\mathcal{S}_{U[1..i]})$ and mark its nodes iff they correspond to the branching nodes of $STree(\mathcal{S}_{U[1..i-1]})$. Note that, due to Fact 2, no edges of $LPT(\mathcal{S}_{U[1..i-1]})$ will be deleted in $LPT(\mathcal{S}_{U[1..i]})$ and only new edges will be added. Hence we can use the NMA data structure on top of this tree.

The next lemma shows how we can efficiently find the new locus of the active point for the updated text $\$_k S_k a$ in the Ukkonen tree.

Lemma 17 *We can compute, in amortized $O(\log \sigma)$ time, the canonical reference to the locus of the active point of $\$_k S_k a$ on the Ukkonen tree, using a data structure which requires $O(n)$ space.*

Proof 19 *Suppose we have constructed the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$ in tandem with the Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$ and $LPT(\mathcal{S}_{U[1..i-1]})$. A node v of $LPT(\mathcal{S}_{U[1..i-1]})$ is marked iff its corresponding node $\overline{v}$ in the Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$ has at least two W-links defined, namely, $\mathsf{w}_c(\overline{v}) = \mathsf{w}_{c'}(\overline{v}) = 1$ with at least two distinct characters $c \neq c'$. This in turn implies that the corresponding node of the (implicitly maintained) DAWG is branching. Every marked node of $LPT(\mathcal{S}_{U[1..i-1]})$ is linked to its corresponding node of the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$ which is also branching by Fact 3 (see also Figure 4.6). We also maintain an NMA data structure on $LPT(\mathcal{S}_{U[1..i-1]})$.*

Given an update operator $U[i] = (k, a)$, we first update the Weiner tree to $STree(\mathcal{T}_{U[1..i]})$. This introduces at most two new hard W-links, one for the new leaf and one for its parent. This means that these edges are also inserted to $LPT(\mathcal{S}_{U[1..i-1]})$ and we then obtain

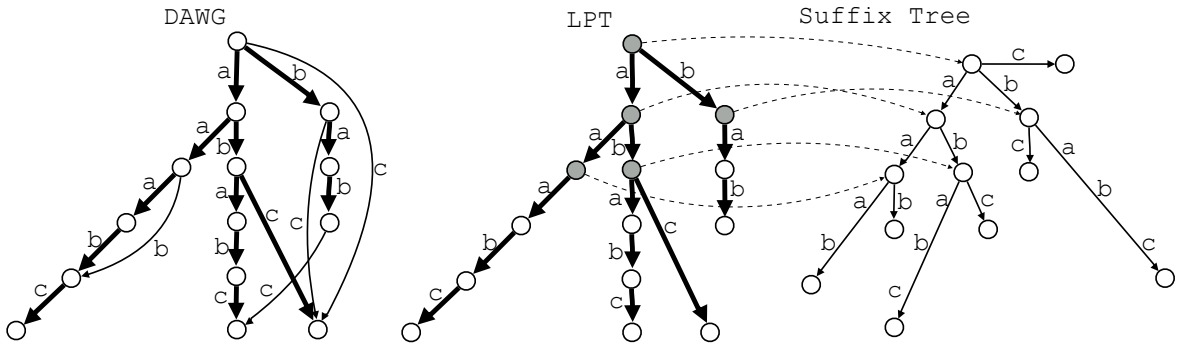

Figure 4.6: Illustration for $DAWG(\mathcal{S}_{U[1..14]})$, $LPT(\mathcal{S}_{U[1..14]})$, and the Ukkonen tree $STree(\mathcal{S}_{U[1..13]})$ before update, where $\mathcal{S}_{U[1..13]} = \{S_1 = \texttt{aaab}, S_2 = \texttt{ababc}, S_3 = \texttt{bab}\}$ and $\mathcal{S}_{U[1..14]} = \{S_1\texttt{c}, S_2, S_3\}$. For simplicity, we here omit the terminate symbols $\$_1$, $\$_2$, and $\$_3$. The bold solid arrows represent the primary edges of $DAWG(\mathcal{S}_{U[1..14]})$, the gray nodes are the marked nodes of $LPT(\mathcal{S}_{U[1..14]})$, and the dashed arrows represent the links between the marked nodes of $LPT(\mathcal{S}_{U[1..14]})$ and the corresponding branching nodes of $STree(\mathcal{S}_{U[1..13]})$. The longest repeating suffix of $S_1\texttt{c}$ in $\mathcal{S}_{U[1..14]}$ is abc, and hence we perform an NMA query from node abc on $LPT(\mathcal{S}_{U[1..14]})$, obtaining node ab. We then access the suffix tree node ab using the link from $LPT(\mathcal{S}_{U[1..14]})$, and obtain the canonical reference $(\texttt{ab}, \texttt{c}, 1)$ to abc on the Ukkonen tree $STree(\mathcal{S}_{U[1..13]})$ before update.

$LPT(\mathcal{S}_{U[1..i]})$. Because of these new edges, at most two DAWG non-branching nodes can become branching. We mark their corresponding nodes in $LPT(\mathcal{S}_{U[1..i-1]})$, and link them to the corresponding Ukkonen tree nodes only after *we have built the updated Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$. This is because the corresponding nodes of $STree(\mathcal{S}_{U[1..i-1]})$ before the update are still non-branching (see Fact 3).*

Let $\overline{y}$ be the insertion point of the leaf $aT_k\$_k$ in the Weiner tree which is the longest repeating prefix of $aT_k\$_k$ in the right-to-left text collection $\mathcal{T}_{U[1..i]}$. By the definition of $LPT(\mathcal{S}_{U[1..i]})$, there is a node in $LPT(\mathcal{S}_{U[1..i]})$ which represents y. We conduct an NMA query from y on $LPT(\mathcal{S}_{U[1..i]})$, and let v be the NMA of y. Let $\ell = |y| - |v|$, and let c be the label of the first edge in the path from v to y. We move from v to its corresponding node x in the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$. Then, (x, c, ℓ) is a reference to the insertion point of the shortest Type-2 suffix of $\$_kS_ka$. Since v is the NMA of y in

$LPT(\mathcal{S}_{U[1..i]})$, and since updating $\$_k S_k$ to $\$_k S_k a$ does not explicitly insert any suffix of $\$_k S_k a$ that is shorter than the longest repeating suffix of $\$_k S_k a$ in $\mathcal{S}_{U[1..i]}$, this reference is canonical by Fact 3.

Clearly the total size of the above data structures is linear in the total length n of the texts in the final text collection $\mathcal{S}$. We analyze the time complexity. We can find the insertion point y of the new leaf in the Weiner tree in amortized $O(\log \sigma)$ time due to Theorem 8. Using the link from the node y in $LPT(\mathcal{S}_{U[1..i]})$, the corresponding node in the Ukkonen tree $STree(\mathcal{S}_{U[1..i-1]})$ can be found in $O(1)$ time. Updating $LPT(\mathcal{S}_{U[1..i-1]})$ to $LPT(\mathcal{S}_{U[1..i]})$ takes $O(\log \sigma)$ amortized time. Inserting a new node and querying an NMA from a given node takes amortized $O(1)$ time. We can link a new marked node of $LPT(\mathcal{S}_{U[1..i]})$ to the corresponding new branching node of $STree(\mathcal{S}_{U1..i})$ in $O(1)$ time, since it is easy to remember this new branching node when updating $STree(\mathcal{S}_{U[1..i-1]})$ to $STree(\mathcal{S}_{U[1..i]})$. Hence, the total amortized bound is $O(\log \sigma)$. ∎

Let w and w' denote the strings that are represented by the loci of the insertion points of the shortest and longest new leaves w.r.t. the update operator $U[i] = (k, a)$. Let $q = |w'| - |w| + 1$ be the number of new leaves to be inserted in the Ukkonen tree. Our backward approach terminates the ith update after inserting the qth new leaf. How do we compute this value q? If (x, c, ℓ) is the canonical reference to the locus for w, then $|w| = |x| + \ell$, and hence what remains is how to compute $|w'|$. We note that w' is the longest suffix of $\$_k S_k$ which has at least one more occurrence in $\mathcal{S}_{U[1..i]}$ immediately followed by another character $b \neq a$. This is because any longer suffix of $\$_k S_k$ is immediately followed only by a, and will thus correspond to existing leaves in the updated Ukkonen tree. These two occurrences of w' must be immediately preceded by distinct characters, say c and d, in the left-to-right text collection $\mathcal{S}_{U[1..i]}$ since otherwise there will be a longer suffix of $\$_k S_k$ which has at least one more occurrence in $\mathcal{S}_{U[1..i]}$, a contradiction. Also, $\overline{w'}c$ and $\overline{w'}d$ occur in the right-to-left text collection $\mathcal{T}_{U[1..i-1]}$ before the ith update. Thus, $\overline{w'}$ is represented by an explicit node in the

Weiner tree $STree(\mathcal{T}_{U[1..i-1]})$. Since this node is on the path from the leaf for $T_k\$_k$ to the root of the Weiner tree, and since it is the deepest node with the hard W-link for character a, we visit this node during the update of the Weiner tree. Hence, we can compute $|w'|$ in $O(\log \sigma)$ amortized time by the aid of the Weiner tree.

The cost to trace the suffix link chains in this backward approach is exactly the same as that in the forward approach. Hence, the total cost is for suffix link chain traversals is $O(n \log \sigma)$ for all $1 \leq i \leq n$ by Lemma 16[1].

The lower bound of Lemma 12 also applies to this backward approach. Hence, we do not maintain the leaf ownerships, and we label the edges leading to the leaves only with their first characters.

We have shown the following:

Theorem 11 *Given a fully-online sequence U of n update operators for a collection of K left-to-right texts $\mathcal{S}$, our backward version of Ukkonen's algorithm can update the suffix tree in a total of $O(n \log \sigma)$ time and $O(n)$ space.*

4.5 Conclusions of Chapter 4

In this chapter, we presented construction algorithms of suffix trees and DAWGs in fully-online manner. Firstly, we showed fully-online construction of suffix trees in right-to-left manner. Secondly, we showed that fully-online construction of DAWGs can be simulated by the right-to-left fully-online suffix tree construction. Finally, we showed that we can fully-online construction of suffix trees with the aid of the fully-online version of DAWGs (right-to-left suffix trees). The time complexity of these algorithms is amortized $O(\log \sigma)$ time per characters.

[1] In the preliminary versions [63] of this capter, a simplified version of the *suffix tree oracle* [33] was used to obtain the same bound. However, we do not need it any more due to our amortization argument of Lemma 16.

Chapter 5

Linear-size Compact Directed Acyclic Word Graphs

In this chapter, we propose a novel approach to combine *compact directed acyclic word graphs* (*CDAWGs*) and grammar-based compression. This leads us to an efficient self-index, called *Linear-size CDAWGs* (*L-CDAWGs*), which can be represented with $O(\tilde{e}_T \log n)$ bits of space allowing for $O(\log n)$-time random and $O(1)$-time sequential accesses to edge labels, and $O(m \log \sigma + occ)$-time pattern matching. Here, $\tilde{e}_T$ is the number of all extensions of maximal repeats in T, n and m are respectively the lengths of the text T and a given pattern, σ is the alphabet size, and occ is the number of occurrences of the pattern in T. The repetitiveness measure $\tilde{e}_T$ is known to be much smaller than the text length n for highly repetitive text. For constant alphabets, our L-CDAWGs achieve $O(m + occ)$ pattern matching time with $O(e^r_T \log n)$ bits of space, which improves the pattern matching time of Belazzougui et al.'s run-length BWT-CDAWGs by a factor of $\log \log n$, with the same space complexity. Here, e^r_T is the number of right extensions of maximal repeats in T. As a byproduct, our result gives a way of constructing a straight-line program (SLP) of size $O(\tilde{e}_T)$ for a given text T in $O(n + \tilde{e}_T \log \sigma)$ time.

5.1 Background

Text indexing is a fundamental problem in theoretical computer science, where the task is to preprocess a given text so that subsequent pattern matching queries can be answered quickly. It has wide applications such as information retrieval, bioinformatics, and big data analytics [49, 57, 59]. There have been a lot of recent research on *compressed text indexes* [9,18,40,49,52,57,62] that store a text T supporting `extract` and `find` operations in space significantly smaller than the total size n of texts. Operation `extract` returns any substring $T[i..j]$ of the text. Operation `find` returns the list of all *occ* occurrences of a given pattern P in T. For instance, Grossi, Gupta, and Vitter [40] gave a compressed text index based on compressed suffix arrays, which takes $s = nH_k + O(n \log\log n \log\sigma / \log n)$ bits of space and supporting $O(m \log\sigma + \text{polylog}(n))$ pattern match time, where H_k is the k-th order entropy of T and m is the length of the pattern P.

Compression measures for highly repetitive text: Recently, there has been an increasing interest in indexed searches for highly repetitive text collections. Typically, the compression size of such a text can be described in terms of some measure of repetition. The followings are examples of such repetitiveness measures for T:

- the *number g_T of rules in a grammar (SLP)* representing T,
- the *number z_T of phrases in the LZ77 parsing* of T,
- the *number r_T of runs in the Burrows-Wheeler transform* of T, and
- the *number $\tilde{e}_T = e^r_T + e^\ell_T$ of right- and left-extensions of maximal repeats* of T.

Belazzougui *et al.* [9] observed close relationship among these measures. Specifically, the authors empirically observed that all of them showed similar logarithmic growth behavior in $|T|$ on a real biological sequence, and also theoretically showed that both z_T and r_T are upper bounded by $\tilde{e}_T$. These repetitive texts are formed from many repeated

fragments nearly identical. Therefore, one can expect that compressed index based on these measures such as g_T, z_T, r_T, and $\tilde{e}_T$ can effectively capture the redundancy inherent to these highly repetitive texts than conventional entropy-based compressed indexes [31, 32, 34, 38, 57, 60, 61] and succinct indexes [5, 41, 43, 54, 66].

Related work There has been extensive research on a family of *repetition-aware indexes* [8–11, 18, 28, 36, 49, 51, 52] since the seminal work by Claude and Navarro [18]. They proposed the first compressed self-index based on grammars, which takes $s = g \log n + O(g \log g)$ bits supporting $O((m^2 + h(m + occ)) \log g)$ pattern match time, where $g = g_T$ and h are respectively the size and height of a grammar. Kreft and Navarro [49] gave the first compressed self-index based on LZ77, which takes $s = 3z \log n + 5n \log \sigma + O(z) + o(n)$ bits supporting $O(m^2 d + (m + occ) \log z)$ pattern match time. Here, d is the height of the LZ parsing. Makinen, Navarro, Siren, and Valimaki [52] gave a compressed index based on RLBWT, which takes $s = r \log \sigma \log(2n/r)(1 + o(1)) + O(r \log \sigma \log \log(2n/r)) + O(\sigma \log n)$ bits supporting $O(m\, f(r \log \sigma, n \log \sigma))$ pattern match time, where $f(b, u)$ is the time for a binary searchable dictionary which is $O((\log b)^{0.5})$ and $o((\log \log u)^2)$ for example [52].

Previous approaches: Considering the above results, we notice that in compression ratio, all indexes above achieve good performance depending on the repetitive measures, while in terms of operation time, most of them except the RLBWT-based one [52] have quadratic dependency in pattern size m. Hence, a challenge here is to develop repetition-aware text indexes to achieve good compression ratio for highly repetitive texts in terms of repetition measures, while supporting faster `extract` and `find` operations. Belazzougui *et al.* [9] proposed a repetition-aware index which combines *CDAWGs* [15, 27] and the run-length encoded BWT [52], to which we refer as *RLBWT-CDAWGs*. For a given text T of the length n and a pattern P of the length m, their index uses $O(e_T^r \log n)$ bits of space and supports `find` operation in $O(m \log \log n + occ)$ time.

Main results: In this chaper, we propose a new repetition-aware index based on combination of CDAWGs and grammar-based compression, called the *Linear-size CDAWG* (L-CDAWG, for short). The L-CDAWG of a text T of length n is a self-index for T which can be stored in $O(\tilde{e}_T \log n)$ bits of space, and support $O(\log n)$-time random access to the text, $O(1)$-time sequential character access from the beginning of each edge label, and $O(m \log \sigma + occ)$-time pattern matching. For constant alphabets, our L-CDAWGs use $O(e_T^r \log n)$ bits of space and support pattern matching in $O(m + occ)$ time, hence improving the pattern matching time of Belazzougui *et al.*'s RLBWT-CDAWGs by a factor of $\log \log n$. We note that RLBWT-CDAWGs use hashing to retrieve the first character of a given edge label, and hence RLBWT-CDAWGs seem to require $O(m \log \log n + occ)$ time for pattern matching even for constant alphabets.

From the context of studies on *suffix indices*, our L-CDAWGs can be seen as a successor of the *linear-size suffix trie* (*LSTries*) by Crochemore *et al.* [23]. The LSTrie is a variant of the suffix tree [26], which need not keep the original text T by elegant scheme of linear time decoding using suffix links and a set of auxiliary nodes. However, it is a challenge to generalize their result for the CDAWG because the paths between a given pair of endpoints are not unique. By combining the idea of LSTries, an SLP-based compression with direct access [13, 37], we successfully devise a text index of $O(\tilde{e}_T \log n)$ bits by improving functionalities of LSTries. As a byproduct, our result gives a way of constructing an SLP of size $O(\tilde{e}_T \log \tilde{e}_T)$ bits of space for a text T. Moreover, since the L-CDAWG of T retains the topology of the original CDAWG for T, the L-CDAWG is a compact representation of all maximal repeats [58] that appear in T.

5.2 Preliminaries

5.2.1 LSTrie

Recently, Crochemore *et al.* [23] proposed a compact variant of a suffix trie, called *linear-size suffix trie* (or LSTrie, for short), denoted $LSTrie(T)$. It is a compacted tree with the topology and the size similar to $STree(T)$, but has no indirect references to a text T (See Fig. 5.1). $LSTrie(T)$ is obtained from $STree(T)$ by adding all nodes v such that their suffix links $slink(v)$ appear also in $STree(T)$. Unlike $STree(T)$, each edge (u, v) of $LSTrie(T)$ stores the first character and the length of the corresponding suffix tree edge label (see Fig. 5.1). Using auxiliary links called the *jump pointers* the following theorem is proved.

Proposition 12 (Crochemore *et al.* [23]) *For a text T of length n, the linear-size suffix trie $LSTrie(T)$ for T can be stored in $O(n \log n)$ bits of space supporting reconstruction of the label of a given edge in $O(\ell)$ time, where ℓ is the length of the edge label.*

Crochemore *et al.*'s method [23] does not regard the order of decoding characters on an edge label. This implies that $LSTrie(T)$ needs $O(\ell)$ worst case time to read any prefix of an edge label of length ℓ. This may cause troubles in some applications including pattern matching. In particular, it does not seem straightforward to match a pattern P against a prefix of the label of an edge e in $O(|P|)$ time when $|P| < |\mathrm{id}lab(e)|$. We will solve these problems in Section 5.3 later.

5.2.2 Straight-line programs

A straight-line program (SLP) is a context-free grammar (CFG) in the Chomsky normal form generating a single string. SLPs are often used in grammar compression algorithms [57].

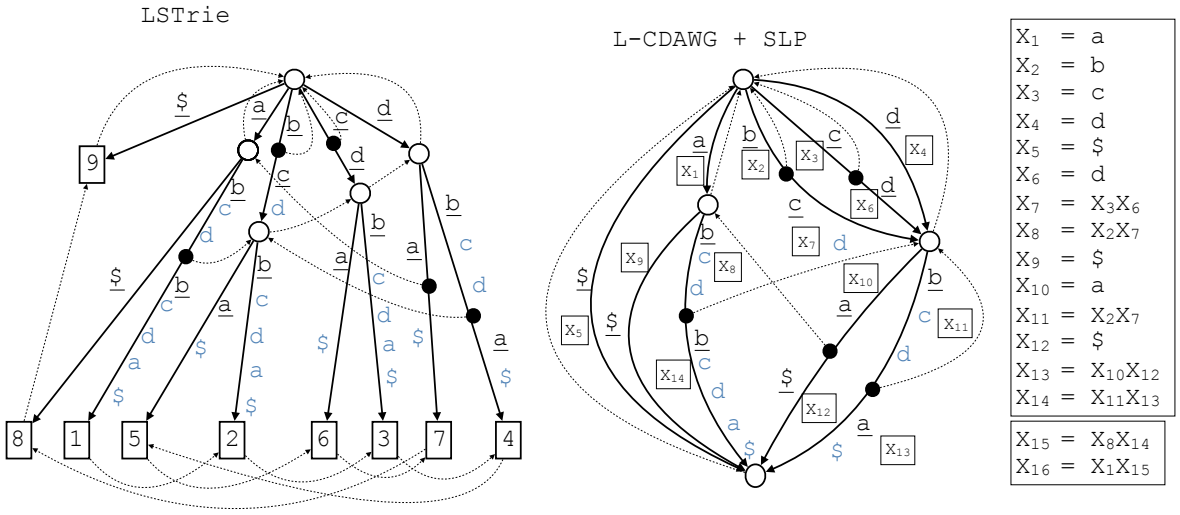

Figure 5.1: Illustration of *LSTrie*(T) and our index structure *L-CDAWG*(T) with *SLP* for text $T =$ `abcdbcda`\$. Solid and broken arrows represent the edges and suffix links, respectively. Underlined and shaded characters attached to each edge are the first (real) and the following (virtual) characters of the original edge label. The expression X_i at the edge indicates the i-th variable of the SLP for T.

Consider an SLP R with n variables. Each production rule is either of form $X \to a$ with $a \in \Sigma$ or $X \to YZ$ without loops. Thus an SLP produces a single string. The *phrase* of each X_i, denoted $\mathcal{F}(X_i)$, is the string that X_i produces. The string defined by SLP R is $\mathcal{F}(X_n)$. We will use the following results.

Proposition 13 (Gasieniec *et al.* [37]) *For an SLP R of size g for a text of length n, there exist a data structure of $O(g \log n)$ bits of space which supports expansion of a prefix of $\mathcal{F}(X_i)$ for any variable X_i in $O(1)$ time per character, and can be constructed in $O(g)$ time.*

Proposition 14 (Bille *et al.* [13]) *For an SLP R of size g representing a text of length n, there exists a data structure of $O(g \log n)$ bits of space which supports to access consecutive m characters at arbitrary position of $\mathcal{F}(X_i)$ for any variable X_i in $O(m + \log n)$ time, and can be constructed in $O(g)$ time.*

5.3 The proposed data structure: L-CDAWG

In this section, we present *the Linear-size CDAWG* (*L-CDAWG*, for short). The L-CDAWG can support CDAWG operations in the same time complexity without holding the original input text and can reduce the space complexity from $O(e_T^r \log n + n \log \sigma)$ bits of space to $O(\tilde{e}_T \log n)$ bits of space, where $\tilde{e}_T = e_T^r + e_T^\ell$ is the number of extensions of maximal repeats. From now on, we assume that an input text T terminates with a unique character \$ which appears nowhere else in T.

5.3.1 Outline

The *Linear-size CDAWG* for a text T of length n, denoted *L-CDAWG*(T), is a DAG whose edges are labeled with single characters. *L-CDAWG*(T) can be obtained from *CDAWG*(T) by the following modifications. From now on, we refer to the original nodes appearing in *CDAWG*(T) as *type-1 nodes*, which are always branching except the sink.

1. First, we add new non-branching nodes, called *type-2 nodes* to *CDAWG*(T). Let $u = value([x])$ for any type-1 node $[x]$ of *CDAWG*(T). If au is a substring of T but the path spelling out au ends in the middle of an edge, then we introduce a type-2 node v representing au. We add the suffix link $u = slink(v)$ as well. Adding type-2 nodes splits an edge into shorter ones. Note that more than one type-2 nodes can be inserted into an edge of *CDAWG*(T).

2. Let (u, x, v) be any edge after all the type-2 nodes are inserted, where $x \in \Sigma^+$. We represent this edge by $e = (u, c, v)$ where c is the first character $c = x[1] \in \Sigma$ of the original label. We also store the original label length $\mathrm{id}slen(e) = |x|$.

3. We will augment *L-CDAWG*(T) with a set of SLP production rules whose non-terminals correspond to edges of *L-CDAWG*(T). The definition and construction

of this SLP will be described later in Section 5.3.3.

If non-branching type-2 nodes are ignored, then the topology of *L-CDAWG*(T) is the same as that of *CDAWG*(T). For ease of explanation, we denote by id$lab(e)$ the original label of edge e. Namely, for any edge $e = (u, c, v)$, id$lab(e) = x$ iff (u, x, v) is the original edge for e.

The following lemma gives an upper bound of the numbers of nodes and edges in *L-CDAWG*(T). Recall that μ_T is the number of maximal repeats in T, e^ℓ_T and e^r_T are respectively the number of left- and right-extensions of maximal repeats in T, and $\tilde{e}_T = e^\ell_T + e^r_T$.

Lemma 18 *For any string T, let L-CDAWG$(T) = (V, E)$, then $|V| = O(\mu_T + e^\ell_T)$ and $|E| = O(\tilde{e}_T)$.*

Proof 20 *Let CDAWG$(T) = (V_0, E_0)$ and CDAWG$(\overline{T}) = (\overline{V_0}, \overline{E_0})$. It is known that $|V_0| = |\overline{V_0}| = \mu_T$, $|E_0| = e^r_T$ and $|\overline{E_0}| = e^\ell_T$ (see [15] and [58]). Let V_1 and V_2 be the set of type-1 and type-2 nodes in L-CDAWG(T), respectively. Clearly, $V_1 \cap V_2 = \emptyset$, $V = V_1 \cup V_2$, and $V_1 = V_0$. Let $[x] \in V_1$ and $u = value([x])$. Note that u is a maximal repeat of T. For any character $a \in \Sigma$ such that au is a substring of T, clearly au is a left-extension of u. By the definition of L-CDAWG(T), it always has a (type-1 or type-2) node which corresponds to au. Hence $|V_2| \leq e^\ell_T$. This implies $|V| = |V_1| + |V_2| = O(\mu_T + e^\ell_T)$. Since each type-2 node is non-branching, clearly $|E| = O(e^r_T + e^\ell_T) = O(\tilde{e}_T)$.* ∎

Corollary 15 *For any string of T over a constant alphabet, $|V| = O(\mu_T + e^r_T)$ and $|E| = O(e^r_T)$, where L-CDAWG$(T) = (V, E)$.*

Proof 21 *It clearly holds that $\mu_T \geq e^\ell_T/\sigma$ and $e^r_T \geq \mu_T$. Thus we have $e^\ell_T \leq \sigma e^r_T$. The corollary follows from Lemma 18 when $\sigma = O(1)$.* ∎

5.3.2 Constructing type-2 nodes and edge suffix links

Lemma 19 *Given CDAWG(T) for a text T, we can compute all type-2 nodes of L-CDAWG(T) in $O(\tilde{e}_T \log \sigma)$ time.*

Proof 22 *We create a copy G of CDAWG(T). For each edge (u, x, v) of CDAWG(T), we compute node $u' = slink(u)$ and the path Q that spells out x from u'. The number of type-1 nodes in this path Q is equal to the number of type-2 nodes that need to be inserted on edge (u, x, v), and hence we insert these nodes to G. After the above operation is done for all edges, G contains all type-2 nodes of L-CDAWG(T). Since there always exists such a path Q, to find Q it suffices to check the first characters of out-going edges. Hence we need only $O(\log \sigma)$ time for each node in Q. Overall, it takes $O(\tilde{e}_T \log \sigma)$ time.* ∎

The above lemma also indicates the notion of the following *edge suffix links* in *L-CDAWG*(T) which are virtual links, and will not be actually created in the construction.

Definition 1 (Edge suffix links) For any edge e with id$slen(e) \geq 2$, id$e$-$suf(e) = (e_1, \ldots, e_k)$ is the path, namely a list of edges, from $e_1.hi = slink(e.hi)$ to $e_k.lo$ that can be reachable from $e_1.hi$ by scanning id$lab(e)$.

Edge suffix links have the following properties.

Lemma 20 *For any edge e such that* id$slen(e) \geq 2$ *and its edge suffix link* id$e$-$suf(e) = (e_1, \ldots, e_k)$, *(1) both $e_1.hi$ and $e_k.lo$ are type-1 nodes, and (2) all nodes in the path $e_1.lo = e_2.hi, \ldots, e_{k-1}.lo = e_k.hi$ are type-2 nodes.*

Proof 23 *From the definition of edge suffix links, we have $e_1.hi = slink(e.hi)$ and the path from $e_1.hi$ to $e_k.lo$ spells out* id$lab(e)$. *(1) By the definitions of type-2 nodes and edge suffix links, $e_1.hi$ is always of type-1. Hence it suffices to show that $e_k.lo$ is of*

type-1. There are two cases: (a) If e.lo is a type-2 node, then by the definition of type-2 nodes, $e_k.lo$ must be the node pointed by slink(e.lo). Therefore, $e_k.lo$ is a type-1 node. (b) If e.lo is a type-1 node, then let ax be the shortest string represented by e.hi with $a \in \Sigma$ and $x \in \Sigma^*$. Then, string $x \cdot \mathrm{id}lab(e)$ is spelled out by a path from the source to $e_1.hi, \ldots, e_k.lo$, where either $e_k.lo = e.lo$ or $e_k.lo = slink(e.lo)$. Since e.lo is of type-1, slink(e.lo) is also of type-1. (2) If there is a type-1 node u in the path $e_2.hi, \ldots, e_{k-1}.lo$, then there has to be a (type-1 or type-2) node v between e.hi and e.lo, a contradiction. ∎

Lemma 20 says that the label of any edge $e = (u, c, v)$ with $\mathrm{id}slen(e) \geq 2$ can be represented by a path $p = (e_1, \ldots, e_k) = \mathrm{id}e\text{-}suf(e)$. In addition, since the path p includes type-1 nodes only at the end points and since type-2 nodes are non-branching, p is uniquely determined by a pair of $(slink(u), c)$. We can compute all edges $e_i \in p$ for $1 \leq i \leq k$ in $O(k + \log \sigma)$ per query, as follows. Firstly, we compute $p.hi = slink(u)$ and then select the out-going edge e_1 starting with the character c in $O(\log \sigma)$ time. Next, we blindly scan the downward path from e_1 while the lower end of the current edge e_i has type-2. This scanning terminates when we reach an edge e_k such that $e_k.lo$ is of type-1.

5.3.3 Construction of the SLP for L-CDAWG

We give an SLP of size $O(\tilde{e}_T)$ which represents T and all edge labels of $L = L\text{-}CDAWG(T)$ based on the jump links.

Jumping from an edge to a path: First, we define *jump links*, by which we can jump from a given edge e with $\mathrm{id}slen(e) \geq 2$ to the path consisting of at least two edges, and having the same string label. Although our jump link is based on that of LSTries [23], we need a new definition since a path in $CDAWG(T)$ (and hence in $L\text{-}CDAWG(T)$) cannot be uniquely determined by a pair of nodes, unlike $STree(T)$ (or $LSTrie(T)$).

Definition 2 (Jump links) For an edge e with id$slen(e) \geq 2$ and id$e$-$suf(e) = (e_1, \ldots, e_k)$, id$jump(e)$ is recursively defined as follows:

1. id$jump(e) :=$ id$jump(e_1)$ if $k = 1$ (thus id$e$-$suf(e) = (e_1)$), and
2. id$jump(e) := (e_1, \ldots, e_k)$ if $k \geq 2$.

Note that id$lab(e)$ equals id$lab(e_1) \cdots$ id$lab(e_k)$ for $jump(e) = (e_1, \ldots, e_k)$.

Lemma 21 *For any edge e with $slen(e) \geq 2$, id$jump(e)$ consists of at least two edges.*

Proof 24 *Assume on the contrary that id$jump(e) = e'$ for some edge e'. This implies id$slen(e') \geq 2$. By definition, $e'.hi$ is a proper suffix of $e.hi$, namely, there exists an integer $k \geq 1$ such that $slink^k(e.hi) = e'.hi$. For any character c which appears in T, there is a (type-1 or type-2) node which represents c as a child of the source of L-CDAWG(T). This implies that there is an out-going edge e'' of length 1 from the source representing the first character of $e.hi$. This contradicts that id$jump(e)$ only contains a single edge e' with id$slen(e') \geq 2$.* ∎

Theorem 16 *For a given L-CDAWG(T), there is an algorithm that computes all jump links in $O(\tilde{e}_T \log \sigma)$ time.*

Proof 25 *We explain how to obtain $jump(e)$ for an edge e with $slen(e) \geq 2$. For all edge e with $slen(e) \geq 2$, we manage a pointer to the first edge e' of $jump(e)$ by $P[e] = e'$. We initially set $P[e] = \epsilon$ for all e. For all nodes e with $slen(e) \geq 2$, let u be an outgoing edge of $slink(e.hi)$ with the same label character of e. We check whether $P[e] = \epsilon$ and, if so, we recursively compute $P[u]$, and then set $P[e] = P[u]$. In this way all $P[e]$ can be computed in $O(\tilde{e}_T \log \sigma)$ time in total, where the $\log \sigma$ is needed for selecting the out going edge. From Lemma 20, since there does not exist branching edge on each jump link, $jump(e)$ can be easily obtained from $P[e]$ by traversing the path until encountered a type-1 node.* ∎

An SLP for the L-CDAWG: We build an SLP which represents all edge labels in *L-CDAWG*$(T) = (V, E)$ based on jump links. For each edge e, let $X(e)$ denote the variable which generates the string label id$lab(e)$. Let $E = \{e_1, \ldots, e_s\}$. For any $e_i \in E$ with id$slen(e_i) = 1$, we construct a production $X(e_i) \to c$ where $c \in \Sigma$ is the label. For any $e_i \in E$ with id$slen(e_i) \geq 2$, let id$jump(e_i) = (e'_1, \ldots, e'_k)$. We construct productions $X(e_i) \to X(e'_1)Y_1$, $Y_1 \to X(e'_2)Y_2$, $\ldots$, $Y_{k-3} \to X(e'_{k-2})Y_{k-2}$, and $Y_{k-2} \to X(e'_{k-1})X(e'_k)$. We call a production whose left-hand size is Y_i an *intermediate* production. It is clear that $X(e_i)$ generates id$lab(e)$ and we introduced $k-1$ productions. If there is another edge e_j $(i \neq j)$ such that id$jump(e_j) = (e'_1, \ldots, e'_k)$, then we construct a new production $X(e_j) \to X(e'_1)Y_1$ and reuse the other productions. Let p be the path that spells out the text T. We create productions which generates T using the same technique as above for this path p. Overall, the total number of intermediate productions is linear in the number of type-2 nodes in *L-CDAWG*(T). Since there are $O(|E|)$ non-intermediate productions, this SLP consists of $O(\tilde{e}_T)$ productions.

Now, we have the main result of this subsection.

Theorem 17 *For a given L-CDAWG*(T)*, there is an algorithm that constructs an SLP which represents all edge labels in* $O(\tilde{e}_T \log \sigma)$ *time.*

Proof 26 *By the above algorithm, if jump links are computed, we can obtain an SLP which represents all edge labels in* $O(\tilde{e}_T)$ *time. From Theorem 16, we can compute all jump links in* $O(\tilde{e}_T \log \sigma)$ *times. Overall, the total time of this algorithm is* $O(\tilde{e}_T \log \sigma)$*.*

∎

Fig. 5.1 shows *LSTrie*(T) and *L-CDAWG*(T) enhanced with the SLP for string $T = abcdbcda\$$.

We associate to each edge label the corresponding variable of the SLP. By applying algorithms of Gasieniec *et al.* [37] (in Proposition 13) and Bille *et al.* [13] (in Proposition 14), we can show the following theorems.

Theorem 18 *For a text T, L-CDAWG(T) can support pattern matching for a pattern P of length m in $O(m \log \sigma + occ)$ time.*

Proof 27 *From Proposition 13, any consecutive m characters from the beginning of an edge in L-CDAWG(T) can be sequentially read in $O(m)$ time. CDAWG(T) can support pattern matching by traversing the path from the source with P in $O(m \log \sigma + occ)$ time [15]. Since L-CDAWG(T) contains the topology of CDAWG(T), it can also support pattern matching in $O(m \log \sigma + occ)$ time.* ∎

Theorem 19 *For a text T of length n, L-CDAWG(T) has an SLP that derives T. In addition, we can read any substring $T[i..i+m]$ can be read in $O(m + \log n)$ time.*

Proof 28 *The text T of L-CDAWG(T) is represented by the longest path p from the source to the sink. Remembering p makes it possible to read any position of T by using the Proposition 14.* ∎

5.3.4 The main result

It is known that for a given string T of length n over an integer alphabet of size $n^{O(1)}$, *CDAWG*(T) can be constructed in $O(n)$ time [55]. Combining this with the preceding discussions, we obtain the main result of this chapter.

Theorem 20 *For a text T of length n, L-CDAWG(T) supports pattern matching in $O(m \log \sigma + occ)$ time for a given pattern of length m and substring extraction in $O(m + \log n)$ time for any substring of length m, and can be stored in $O(\tilde{e}_T \log n)$ bits of space (or $O(\tilde{e}_T)$ words of space). If CDAWG(T) is already constructed, then L-CDAWG(T) can be constructed in $O(\tilde{e}_T \log \sigma)$ total time. If T is given as input, then L-CDAWG(T) can be constructed in $O(n + \tilde{e}_T \log \sigma)$ total time for integer alphabets of size $n^{O(1)}$. After L-CDAWG(T) has been constructed, the input string T can be discarded.*

5.4 Conclusions of Chapter 5

In this chapter, we presented the Linear-size CDAWG that is a new text index for repetitive texts. The Linear-size CDAWG for a text T, denoted by $L\text{-}CDAWG(T)$, can be obtained by combining the CDAWG for T and grammar-based compression. $L\text{-}CDAWG(T)$ takes $O(\tilde{e}_T \log n)$ bits of space and supports $O(m \log \sigma + occ)$ time pattern matching, where $\tilde{e}_T$ is the number of right- and left- extensions of maximal repeats in T, n and m are respectively the lengths of the text T and a given pattern. We also proposed a construction algorithm of Linear-size CDAWGs in $O(n + \tilde{e}_T \log \sigma)$ time.

Chapter 6

Conclusions and Future Work

In this thesis, we studied efficient index construction for multiple and repetitive texts. In particular, we consider three problems: supporting fast construction and queries, online construction, and compression of space.

6.1 Summary of the results

We summarize the results of this thesis as follows. In Chapter 3, we presented a new data structure called the *packed compact trie* (*packed c-trie*), which stores a set S of k strings of total length n in $n \log \sigma + O(k \log n)$ bits of space, where σ is the size of an alphabet. Given a string of length m, we show that our packed c-tries support pattern matching queries and insert/delete operations in $O(\frac{m}{\alpha} f(k, n))$ worst-case time and in $O(\frac{m}{\alpha} + f(k, n))$ expected time. We discussed applications of packed c-tries to online sparse suffix tree construction and LZD factorization.

In Chapter 4, the main contribution of this chapter is an $O(n \log \sigma)$-time algorithm to maintain the suffix tree for a text collection in the left-to-right fully-online setting, where n and σ are the total text length and the alphabet size, respectively. We also propose construction algorithms for suffix trees in right-to-left fully-online setting. It

coincides with fully-online construction for DAWGs. The key was a non-trivial use of the right-to-left suffix tree construction algorithm.

In Chapter 5, we presented a new repetition-aware data structure called Linear-size CDAWGs. *L-CDAWG*(T) takes linear space in the number of the left- and right-extensions of the maximal repeats in T, which is known to be small for highly repetitive strings. The key idea is using a small SLP induced from edge-suffix links. This SLP is repetition-aware, i.e., its size is linear in the number of left- and right-extensions of the maximal repeats in T. We also showed how to efficiently construct *L-CDAWG*(T).

Overall, in this thesis, we considered three problems to have an ideal text index for a massive amount of text data, that is, a fast and small index which can be constructed in fully-online manner. Firstly, we proposed an index that supports fast queries. Secondly, we proposed an index for multiple texts which can be constructed in fully-online manner. Finally, we proposed an index for repetitive texts that can be stored in small space. These results can improve the utility of suffix data structures such as suffix trees and CDAWGs, and these are a first step towards an index that has all of three features.

6.2 Future work

For future work, we would like to combine these results for more efficient indexes.

- One of the interesting challenge is to combine the results of Chapter 4 with the results of Chapter 5. For fully-online construction algorithms of CDAWGs, it may be possible by using fully-online construction for suffix trees which we proposed. More challenging task is to propose online and fully-online construction algorithm for Linear-size CDAWGs.

- It is also a future work to combine the results of Chapter 3 and Chapter 4 to propose a faster fully-online construction algorithm. The fully-online algorithm

of suffix trees relies on NMA queries and traverse of suffix links. However, results of Chapter 3 is to speed up traverse of tree branches. Therefore, it is difficult to simply adapt the result of Chapter 3 is to speed up for fully-online construction.

Acknowledgement

I would like to express my special thanks to my supervisor Hiroki Arimura. Without his guidance and persistent help this thesis would not have been possible. He also advise me how to behave as a researcher. I would like to express thanks to associate professor Takuya Kida. He always gave advice to me when I had a trouble. I would also like to thank Professor Shin-ichi Minato and Professor Thomas Zeugmann for their comments on this thesis. I am grateful to associate professor Shunsuke Inenaga for helpful discussions and insightful comments. Our papers has improved very much with his help. I can not finish writing this thesis without his kind support.

My co-authors, Hiroki Arimura, Dany Breslauer, Diptarama, Yuta Fujishige, Keisuke Goto, Shunsuke Inenaga, Yuto Nakashima, and Kunihiko Sadakane, always kindly supported during writing our papers. I would be grateful to Ms. Manabe and the members of our laboratory. Thanks to them, my laboratory life has been enriched. Especially, Ms. Manabe greatly helped me with business trip procedures and document procedures. Finally, I would like to thank my parents for supporting my student life.

Bibliography

[1] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.

[2] Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2000.

[3] Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):13:1–13:40, 2007.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS 2002, June 3-5, Madison, Wisconsin, USA*, pages 1–16. ACM, 2002.

[5] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 680–689. SIAM, 2007.

[6] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38 – 72, 2002.

[7] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 159–172. Springer, 2010.

[8] Djamal Belazzougui and Fabio Cunial. Representing the suffix tree with the CDAWG. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:13. Schloss Dagstuhl, 2017.

[9] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 26–39. Springer, 2015.

[10] Djamal Belazzougui and Gonzalo Navarro. Alphabet-independent compressed text indexing. *ACM Transactions on Algorithms*, 10(4):23:1–23:19, 2014.

[11] Djamal Belazzougui and Gonzalo Navarro. Optimal lower and upper bounds for representing sequences. *ACM Transactions on Algorithms*, 11(4):31:1–31:21, 2015.

[12] Oren Ben-Kiki, Philip Bille, Dany Breslauer, Leszek Gasieniec, Roberto Grossi, and Oren Weimann. Optimal packed string matching. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 423–432. Schloss Dagstuhl, 2011.

[13] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

[14] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automation recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[15] Anselm Blumer, Janet Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[16] Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *Journal of Discrete Algorithms*, 18:32–48, 2013.

[17] Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):21:1–21:29, 2007.

[18] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

[19] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 91–100. ACM, 2004.

[20] Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[21] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[22] Maxime Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45(1):63 – 86, 1986.

[23] Maxime Crochemore, Chiara Epifanio, Roberto Grossi, and Filippo Mignosi. Linear-size suffix tries. *Theoretical Computer Science*, 638:171–178, 2016.

[24] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[25] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

[26] Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2002.

[27] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching, 8th Annual Symposium, CPM 97, Aarhus, Denmark, June 30 - July 2, 1997, Proceedings*, volume 1264 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1997.

[28] Huy Hoang Do, Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Fast relative lempel-ziv self-index for similar sequences. *Theoretical Computer Science*, 532:14–30, 2014.

[29] Paolo Ferragina and Roberto Grossi. Improved dynamic text indexing. *Journal of Algorithms*, 31(2):291–319, 1999.

[30] Paolo Ferragina and Roberto Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[31] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[32] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20:1–20:24, 2007.

[33] Johannes Fischer and Paweł Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2015.

[34] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[35] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993.

[36] Travis Gagie, PawełGawrychowski, and Simon J. Puglisi. Approximate pattern matching in lz77-compressed texts. *Journal of Discrete Algorithms*, 32(C):64–68, 2015.

[37] Leszek Gasieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant. Real-time traversal in grammar-based compressed files. In *2005 Data Compression Conference (DCC 2005), 29-31 March 2005, Snowbird, UT, USA*, page 458. IEEE, 2005.

[38] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 368–373. SIAM, 2006.

[39] Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 219–230. Springer, 2015.

[40] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, pages 841–850. SIAM, 2003.

[41] Roberto Grossi, Alessio Orlandi, and Rajeev Raman. *Optimal Trade-Offs for Succinct String Indexes*, volume 6198 of *Lecture Notes in Computer Science*, pages 678–689. Springer, 2010.

[42] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[43] Meng He and J. Ian Munro. *Succinct Representations of Dynamic Strings*, volume 6393 of *Lecture Notes in Computer Science*, pages 334–346. Springer, 2010.

[44] Wing-Kai Hon, Tak-Wah Lam, Rahul Shah, Siu-Lung Tam, and JeffreyScott Vitter. Succinct index for dynamic dictionary matching. In *Algorithms and Computation, 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings*, volume 5878 of *Lecture Notes in Computer Science*, pages 1034–1043. Springer, 2009.

[45] Shunsuke Inenaga and Masayuki Takeda. On-line linear-time construction of word suffix trees. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2006.

[46] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 71(4):969–988, 2015.

[47] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Computing and Combinatorics, Second Annual International Conference, COCOON '96, Hong Kong, June 17-19, 1996, Proceedings*, volume 1090 of *Lecture Notes in Computer Science*, pages 219–230. Springer, 1996.

[48] Eamonn Keogh, Stefano Lonardi, and Bill 'Yuan-chi' Chiu. Finding surprising patterns in a time series database in linear time and space. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23-26, 2002, Edmonton, Alberta, Canada*, pages 550–556. ACM, 2002.

[49] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[50] M. Lothaire. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005.

[51] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[52] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[53] Donald R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[54] J.Ian Munro, Venkatesh Raman, and S.Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[55] Kazuyuki Narisawa, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient computation of substring equivalence classes with suffix arrays. *Algorithmica*, 79(2):291–318, 2017.

[56] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[57] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2:1–2:61, 2007.

[58] Mathieu Raffinot. On maximal repeats in strings. *Information Processing Letters*, 80(3):165–169, 2001.

[59] S. Rao Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 631–638. IEEE, 1995.

[60] Luís M. S. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53:1–53:34, 2011.

[61] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[62] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Improved ESP-index: a practical self-index for highly repetitive texts. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 338–350. Springer, 2014.

[63] Takuya Takagi, Shunsuke Inenaga, and Hiroki Arimura. Fully-online construction of suffix trees for multiple texts. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54

of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:13. Schloss Dagstuhl, 2016.

[64] Takashi Uemura and Hiroki Arimura. Sparse and truncated suffix trees on variable-length codes. In *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, volume 6661 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2011.

[65] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[66] Sebastiano Vigna. Quasi-succinct indices. In *Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, February 4-8, 2013*, pages 83–92. ACM, 2013.

[67] Yilun Wang, Yu Zheng, and Yexiang Xue. Travel time estimation of a path using sparse trajectories. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 25–34. ACM, 2014.

[68] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE, 1973.

[69] Jeffery Westbrook. Fast incremental planarity testing. In *Automata, Languages and Programming, 19th International Colloquium, ICALP 1992, Vienna, Austria, July 13-17, 1992, Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 342–353. Springer, 1992.

[70] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17:81–84, 1983.

[71] Dan E. Willard. New trie data sturucture which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):379–394, 1984.

[72] Yu Zheng. Trajectory data mining: An overview. *ACM Transactions on Intelligent Systems and Technology*, 6(3):29:1–29:41, 2015.

[73] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.