Combinatorial Approaches for Compact String Indexing and Efficient Pattern Discovery

藤重, 雄大

https://doi.org/10.15017/4060184

Combinatorial Approaches for Compact String Indexing and Efficient Pattern Discovery

Yuta Fujishige

January, 2020

Abstract

With recent improvements in observational technology and developments of the Internet, we can obtain large amounts of data such as meteorological data, web data, and biological data. It is an urgent task to develop technologies for extracting useful and valuable information from such data that will drive decisions or future actions. The difficulty in dealing with the rapidly growing data is not only because they are simply vast, but also because they are often "unstructured" and "non-uniform". Current Database technologies are developed on the theory of relational algebra, and are able to handle structured data (i.e. relations) that can be represented as tables, but have difficulty in handling such unstructured data.

To overcome the difficulty, we consider unstructured data as sequences of character symbols, i.e. strings, and investigate foundational technologies for efficient processing of string data, based on the theory of Combinatorics on Strings. The aims of this thesis are: (A) time- and space-efficient string indexing structures and (B) development of efficient pattern discovery algorithms, by exploiting combinatorial properties of strings.

(A) We focus on the well-known string indexing structures: Suffix Trees and DAWGs. We present the first linear-time algorithm to construct DAWG for strings over an integer alphabet. We next consider the pattern matching for short patterns and propose an index structure named the *truncated DAWG*, which is obtained from DAWG by deleting some nodes and edges. We show that the truncated DAWGs can be represented in small space. As an application of the truncated DAWG, we show an algorithm for the length-constrained minimal absent words problem. Then, we address the generic words problem: *Given a collection D of strings, build a data structure which takes as input a string p and a threshold $d > 0$ and enumerates all superstrings of p occurring in at least d strings of D.* Based on the suffix trees we propose an $O(n \log |D|)$-size data structure that solves the problem using $O(|p| + o \cdot \log\log |D|)$ time where o is the output size. The proposed data structure outperforms the previous work by Nishimoto, et al. both in the query time and in the space requirement.

(B) We address the problem of enumerating *combinatorial objects* occurring in a given string. The combinatorial objects we consider are the *gapped palindromes* and the *maximal repetitions*. We propose an efficient algorithm to enumerate the length-constrained gapped palindromes in an online manner. Then, we consider enumerating the maximal repetition in a string compressed by the run-length encoding (RLE). We show a new upper bound on the number of maximal repetitions in a string in terms of the compressed input size. We also present an algorithm for enumerating all maximal repetitions in a RLE compressed string that runs in almost linear time with respect to the RLE-compressed size.

Acknowledgments

I would like to express my gratitude to everyone who supported my research life at Kyushu University. First of all, I am deeply grateful to Professor Masayuki Takeda who is my supervisor and the committee chair of my thesis. He taught depth of research, attitude toward research, and so on to me. I would also like to express my appreciation to Professor Eiji Takimoto and Associate Professor Daisuke Ikeda, who are the members of the committee of my thesis. I also thank all of those in Department of Informatics, Kyushu University, for their generous support.

I would also express my appreciation to Associate Professor Hideo Bannai, Associate Professor Shunsuke Inenaga and Assistant Professor Yuto Nakashima. They taught how to research and gave knowledge and many ideas to me. I am deeply grateful to Professor Eiji Takimoto and Associate Professor Kohei Hatano. They gave many fruitful comments to me in weekly seminar.

This research was partly supported by JSPS (Japan Society for the Promotion of Science). The results in the thesis were partially published in the Proc. of IWOCA'16, the Proc. of MFCS'16, the Proc. of ISAAC'17, the Proc. of SPIRE'18, and the Proc. of ISAAC'19. I am thankful for all editors, committees, anonymous referees, and publishers.

I would like to thank my appreciation to Dr. Juha Kärkkäinen and Dr. Simon J. Puglisi. I enjoyed the discussion with them in my visit to Department of Computer Science, University of Helsinki.

Last, but not least, I really thank my family for their support.

Contents

Chapter 1

Introduction

With recent improvements in observational technology and developments of the Internet, we can obtain large amounts of data every day, such as meteorological data, web data, biological data, and so on. It is an urgent task to develop technologies for extracting useful and valuable information from such data that will drive decisions or future actions. The difficulty in dealing with the rapidly growing data is not only because they are simply vast, but also because they are often "unstructured" and "non-uniform". Current Database technologies are developed on the theory of relational algebra, and are able to handle structured data (i.e. relations) that can be represented as tables, but have difficulty in handling such unstructured data.

To overcome the difficulty, we consider unstructured data as sequences of character symbols, i.e. *strings*, and aim to establish foundational technologies for efficient processing of string data. The ordinary techniques in Data Structures and Algorithms are not sufficient for efficient string processing: We need novel techniques specialized for manipulating strings based on the theory of *Combinatorics on Strings (or Words)* [73] are required.

Let us consider the *string pattern matching problem*, one of the most important problems in Computer Science, defined as follows: Given a pattern string and a text string, to find all occurrences of the pattern in the text. The first linear-time solution to this problem is the *Knuth-Morris-Pratt (KMP) algorithm* [43]. The linearity is guaranteed by *Periodicity Lemma* [29], a well-known lemma on the periodicity and repetitions, one of the most important characteristic features in strings. The *Crochemore-Perrin algorithm* [24] is also a linear time solution using *only* constant space, whereas the KMP algorithm requires linear space in the pattern length. The correctness is guaranteed by the *Critical Factorization Theorem* [55], an important theorem on the periodicity and repetitions. From these facts, we believe that discovering new combinatorial properties on strings leads to developments of simple and efficient string-processing algorithms.

On the other hand, consider the same problem in the static text and dynamic pattern situation, a traditional keyword search in a non-changing database. An efficient solution is to build a *string indexing structure* allowing a fast search, such as *suffix trees* [75], *suffix arrays* [61] and *directed acyclic word graphs* (*DAWGs*) [12, 19]. String indexing structures are fundamental data structures useful for not only substring search but also for various string processing, such as string pattern discovery, string data compression, and so on, and have been extensively studied. Suffix trees and DAWGs are defined using the equivalence relations $\equiv_L$ and $\equiv_R$, and the equivalence relations are, respectively, defined based on the equality of the sets of beginning positions and the sets of ending positions of substring occurrences. The time and space complexities of suffix trees and DAWGs are guaranteed by the combinatorial properties on the equivalence relations.

The aims of this thesis are: (A) time- and space-efficient construction of string indexing structures and (B) development of efficient pattern discovery algorithms, by exploiting the combinatorial properties on strings.

1.1 Our Problems and Our Contributions

In this section, we state the problems we consider in this thesis and explain motivations and our results for the problems.

1.1.1 (A) Time- and space-efficient construction of indexing structures

As mentioned above, indexing structures are used for various problems. We proposed a faster construction algorithm and more space-saving indexes for some string sets.

(A-1) Constructing full text index DAWG in linear time for integer alphabet

Time complexities for constructing string index data structures depend on the underlying alphabet. For a given string y of length n over an *ordered alphabet* of size σ, the suffix tree [62], the suffix array [61], the DAWG, and the *compact DAWGs* (*CDAWGs*) [13] of y can all be constructed in $O(n \log \sigma)$ time. These immediately lead to $O(n)$-time construction algorithms for a *constant alphabet* of constant size.

We are particularly interested in the computation of string index data structures for input strings of length n over an *integer alphabet* of polynomial size in n. For this situation,

$O(n \log \sigma)$-time construction algorithms for ordered alphabet take $O(n \log n)$ time for integer alphabet. Farach-Colton et al. [28] proposed the first $O(n)$-time suffix tree construction algorithm for integer alphabets. Since the out-edges of every node of the suffix tree constructed by McCreight's [62] and Farach-Colton et al.'s algorithms are lexicographically sorted, and since sorting is an obvious lower-bound for constructing edge-sorted suffix trees, the above-mentioned suffix-tree construction algorithms are optimal for ordered and integer alphabets, respectively. Since the suffix array of y can be easily obtained in $O(n)$ time from the edge-sorted suffix tree of y, suffix arrays can also be constructed in optimal time. In addition, since the edge-sorted suffix tree of y can easily be constructed in $O(n)$ time from the edge-sorted CDAWG of y, and since the edge-sorted CDAWG of y can be constructed in $O(n)$ time from the edge-sorted DAWG of y [13], sorting is also a lower-bound for constructing edge-sorted DAWGs and edge-sorted CDAWGs. Using the technique of Narisawa et al. [66], edge-sorted CDAWGs can be constructed in optimal $O(n)$ time for integer alphabets. On the other hand, the only known algorithm to construct DAWGs was Blumer et al.'s $O(n \log \sigma)$-time online algorithm [12] for ordered alphabets of size σ, which results in $O(n \log n)$-time DAWG construction for integer alphabets.

We close the gap between the upper and lower bounds for DAWG construction, by proposing the first $O(n)$-time algorithm to construct edge-sorted DAWGs for integer alphabets. Our algorithm also computes the suffix links, and can thus be applied to various kinds of string processing problems. Our algorithm builds $DAWG(y)$ for a given string y by transforming the suffix tree of y to $DAWG(y)$. In other words, our algorithm simulates the minimization of the suffix trie of y to $DAWG(y)$ using only $O(n)$ time and space.

A simple modification to our $O(n)$-time DAWG construction algorithm also leads us to the first $O(n)$-time algorithm to construct affix trees for integer alphabets. We remark that the previous best known affix-tree construction algorithm of Maaß [57] requires $O(n \log n)$ time for integer alphabets.

(A-2) Efficient indexing data structure for short patterns

Na et al. [65] proposed k-truncated suffix trees which are the pruned version of suffix trees that require less space than the suffix trees in practice. This can perform the same operation as a suffix tree for a pattern with a length less than or equal to k.

The DAWG of a string y, denoted by $DAWG(y)$, is an edge-labeled directed acyclic graph obtained by merging isomorphic subtrees of the suffix trie of y. It is known that each node in

$DAWG(y)$ represents substrings of y that have the same set of ending positions. On the other hand, $DAWG(y)$ also can be seen as the smallest automaton recognizing all suffixes of y. We can make the smallest automaton recognizing all substrings of length k or less, by minimizing the trie of the substrings of y whose length is less than or equal to k. However, it is difficult to construct such automaton efficiently and sometimes its size does not become small, for example, the size is $\Theta(kn)$ when all characters in y are different from one another.

We propose a new data structure named the *k-truncated DAWG*, obtained from DAWG by removing some nodes and edges. The k-truncated DAWG of y, denoted by $k\text{-}TDAWG(y)$, is a subgraph of $DAWG(y)$ where a node in $DAWG(y)$ is also a node in $k\text{-}TDAWG(y)$ if and only if the length of the shortest string represented by the node in $DAWG(y)$ is k or less. We show that the $k\text{-}TDAWG(y)$ can be stored in $O(\min\{n, k\gamma\})$ space, where n is the length of y and γ is the size of one of the smallest k-attractors of y [42]. We also present an online algorithm that constructs $k\text{-}TDAWG(y)$ in $O(n \log \sigma)$ time and $O(\min\{n, k\gamma\})$ space, where σ is the alphabet size. We modify the online DAWG construction algorithm by Blumer et al. [12] by adding node and edge deletion operations to the algorithm and show that these deletion operations can be performed safely while maintaining $O(\min\{n, k\gamma\})$ working space.

For a string y, it is known that the suffix links of the $DAWG(y)$ coincide with the edges of the suffix tree of y^R [25], where y^R is the reverse string of y. We show that this property also holds between the truncated DAWG of y and the truncated suffix tree of y^R. Moreover, the truncated DAWG of y contains secondary edges, which are not present in the truncated suffix tree of y^R.

As an application of $k\text{-}TDAWG(y)$, we present an algorithm to compute the set $MAW_k(y)$ of all minimal absent words of y whose length is smaller than or equal to k by using $k\text{-}TDAWG(y)$. A string x is said to be a minimal absent word of y if x does *not* occur in y and all proper substrings of x occur in y. Minimal absent words have some applications such as to build phylogeny [14] and pattern matching [20]. Let $MAW(y)$ be the set of minimal absent words of y. Fujishige et al. [32] proposed an algorithm to compute $MAW(y)$ by using $DAWG(y)$ in $O(n + |MAW(y)|)$ time. This problem cannot be solved using the suffix tree of y and its suffix links in the same time and space complexity. In this chapter, we show that $MAW_k(y) = \{x \mid x \in MAW(y), |x| \leq k\}$ can be computed by using k-truncated DAWG in $O(\min\{n, k\gamma\} + |MAW_k(y)|)$ time. Similar to $MAW(y)$, $MAW_k(y)$ cannot be computed using the truncated suffix tree of y with its suffix links in the same time and space complexity.

Last, we check the size of k-truncated DAWGs compared to the size of DAWGs and LZ77 factorization by experiments. We also compare the construction time of DAWGs and truncated DAWGs, and the time to compute MAW_k by using them.

(A-3) Frequent pattern mining for a string set

Frequent substring patterns are often referred to as *generic words*. The generic words mining problem (or the frequent substring pattern mining problem) has a wide variety of applications, e.g., Computational Biology, Text mining, and Text Classification [51, 11, 36]. One interesting variant of the generic words mining problem is the *right maximal generic words problem*, formulated by Kucherov et al. [51]. In this variant, a pattern p is given as additional input, which limits the outputs to the right extensions of p. Moreover, the outputs are limited to the *maximal* ones. Formally, the problem is to preprocess D so that, for any pattern p and any threshold d, all right extensions of p that are d-right maximal can be computed efficiently, where a string w is said to be *d-right maximal* if x occurs in at least d documents but xa occurs in less than d documents for any character a. They presented in [51] an $O(n)$-size data structure which answers queries in $O(|p| + r)$ time, where n is the total length of strings in D and r is the number of outputs. Later, Biswas et al. [11] developed a succinct data structure of size $n \log |\Sigma| + o(n \log |\Sigma|) + O(n)$ bits of space, which answers queries in $O(|p| + \log \log n + r)$ time.

As a generalization, Nishimoto et al. [67] defined the *left-right-maximal generic word problem*. In this problem, all superstrings of p that are d-left-right maximal should be answered, where a string w is said to be *d-left-right maximal* if x has a document frequency $\geq d$ but xa and ax respectively have a document frequency $< d$ for any character a.

One naive solution to this problem is to compute the sets M_d of d-left-right maximal strings for $1 \leq d \leq m$, where m is the number of documents in D and then apply the optimal algorithm of Muthukrishnan [64] for the document listing problem, regarding M_d as input document collection. The query time is $O(|p| + o)$ time, where o is the number of outputs. The space requirement is $O(n^2 \log m)$ since the Muthukrishnan algorithm uses the (generalized) suffix tree of input document collection and the size of suffix tree for M_d can be shown to be $O(n^2/d)$ for every $d = 1, \ldots, m$. The $O(n^2 \log m)$ space requirement is, however, impractical when dealing with a large-scale document collection.

In [67] Nishomoto et al. presented an $O(n \log n)$-space data structure which answers queries in $O(|p| + r \log n + o \log^2 n)$ time, where r is the number of d-right-maximal strings that subsume p as a prefix. The factor $O(r \log n)$ is for computing the d-right-maximal right extensions of p, which are required for computing d-left-right-maximal extensions of p in their method.

We address the left-right-maximal generic word problem and propose an $O(n \log m)$-space data structure with query time $O(|p| + o \log \log m)$. The data structure outperforms the previous work by Nishimoto et al. [67] both in the query time and in the space requirement.

Our method uses the suffix trees of M_d for $d = 1, \dots, m$. For a string set $S = \{w_1, \dots, w_\ell\}$, Usually, "the suffix tree of S" means the suffix tree of $\{w_1\$_1, \dots, w_\ell\$_\ell\}$ with ℓ distinct endmarkers $\$_1, \dots, \$_\ell$, or the suffix tree of $S\$ = \{w_1\$, \dots, w_\ell\$\}$ with a single endmarker \$. In both cases, the size of the suffix tree is proportional to the total length of the strings in S. The total size of suffix trees of $M_d\$$ for $d = 1, \dots, m$ is $O(nm)$, where n is the total length of D. Our idea in reducing the space requirement is to replace the suffix tree of $M_d\$$ with the suffix tree of M_d. Removing the endmarker successfully reduces the $O(nm)$ total size of the suffix trees to $O(n \log m)$, with a small sacrifice of query time.

1.1.2 (B) Efficient pattern discovery

String indexing data structures such as suffix trees and DAWGs can be applied in various problems on strings according to their properties.

(B-1) Computing gapped palindromes

A *palindrome* is a string of form xax^R, where x is a string called the left arm, a is either the empty string or a single character, and x^R is the reversed string of x called the right arm. Finding palindromic substrings in a given string w is a classical problem on string processing. The earliest work on this problem dates back to at least 1970's when Manacher [60] proposed an online algorithm to find all prefix palindromes in w in $O(n)$ time, where n is the length of w. Later, Apostolico et al. [3] pointed out that Manacher's algorithm can be used to find all maximal palindromes in w in $O(n)$ time, where a maximal palindrome is a substring palindrome $w[i..j] = w[i..j]^R$ of w whose arms cannot be further extended based on the same center $\frac{i+j}{2}$.

A natural generalization of palindromes is *gapped palindromes* of form xyx^R, where y is a string of length at least 2 called a *gap*[1]. Finding gapped palindromes has applications in bioinformatics, e.g.; RNA secondary structures called hairpins can be regarded as a kind of gapped palindrome $xy\overline{x}^R$, where $\overline{x}$ represents the complement of x ($\overline{x}$ is obtained by exchanging `A` with `U` and exchanging `C` with `G` in x). The most basic type of gapped palindromes is *g-gapped*

[1] If y is a single character, then xyx^R is a palindrome of odd length. Thus we here assume y is of length at least 2.

palindromes, where $g \geq 2$ is a pre-defined fixed length of the gaps. For three parameters $g_{\min}$, $g_{\max}$, and A such that $2 \leq g_{\min} \leq g_{\max}$ and $A \geq 1$, Kolpakov and Kucherov [45] introduced *length-constrained gapped palindromes* (*LCGPs*) which has arms of length at least A and gaps of length in range $[g_{\min}, g_{\max}]$. This is a natural generalization of g-gapped palindromes with $g_{\min} = g_{\max} = g$ and $A = 1$. Kolpakov and Kucherov [45] proposed an $O(n \log \sigma + L)$-time offline algorithm to find all maximal LCGPs, where L is the number of outputs.

We consider the problems of finding LCGPs in a string in an *online manner*. Namely, our input is a growing string to which new characters can be appended, and each character of the string arrives one by one, from left to right. Let n be the length of the final string w. We propose an online algorithm to compute all maximal LCGPs in w in $O(n(\frac{g_{\max}-g_{\min}}{A} + \log \sigma) + occ)$ time and $O(n)$ space, where occ is the number of the outputs.

To our knowledge, there was no algorithm to compute gapped palindromes online until we proposed this algorithm.

(B-2) Computing maximal repetitions in run length encoded strings

Periodicity and repetitions in strings are one of the most important characteristic features in strings. They have been one of the first objects of study in the field of combinatorics on words [73] and have many theoretical, as well as practical applications, e.g., in bioinformatics [44].

Maximal repetitions are periodically maximal substrings of a string where the smallest period is at most half the length of the substring, i.e., there are at least two consecutive occurrences of the same substring. An $O(n \log n)$ time algorithm for computing all of the maximal repetitions contained in a string of length n, was shown by Main and Lorentz [59], which is optimal for general unordered alphabets, i.e., when only equality comparisons between the letters are allowed. Kolpakov and Kucherov [46] further showed that the number of maximal repetitions is actually $O(n)$, and gave a linear time algorithm for ordered constant size alphabets (and essentially for integer alphabets), to compute all of them. The algorithm was a modification of the algorithm by Main [58], which in turn relies on the Lempel-Ziv 77 (LZ77) factorization [77] of the string, which can be computed in linear time for ordered constant size or integer alphabets [21], but requires $\Omega(n \log \sigma)$ time for general ordered alphabets [47], where σ is the size of the alphabet. Recently, a new characterization of maximal repetitions using Lyndon words was proposed by Bannai et al. [6, 7], which lead to a very simple proof to what was known as the “runs” conjecture, i.e., that the number of maximal repetitions in a given string of length n

is less than n [7]. The characterization also lead to a new linear time algorithm for computing maximal repetitions on ordered constant size and integer alphabets, which does not require the LZ77 factorization, but only on a linear number of *longest common extension queries*. Furthermore, based on this algorithm, the running time for computing all maximal repetitions for general ordered alphabets were subsequently improved to $O(n \log^{2/3} n)$ by Kosolobov [48], $O(n \log \log n)$ by Gawrychowski et al. [35], and $O(n\alpha(n))$ by Crochemore et al. [22], where α denotes the inverse Ackermann function.

We consider the problem of computing all maximal repetitions contained in a string when given the *run-length encoding* (RLE) of the string, which is a well known compressed representation where each maximal substring of the same character is encoded as a pair consisting of the letter and the length of the substring. For example, the run-length encoding of the string aaaabbbaaacc is $(\mathtt{a}, 4)(\mathtt{b}, 3)(\mathtt{a}, 3)(\mathtt{c}, 2)$. The main contributions of the chapter are:

1. an upper bound $m + k - 1$ on the number of maximal repetitions contained in a string, where m is the size of its run-length encoding and k is the number of run-length factors whose exponent is at least 2, and

2. an $O(m\mathcal{A}(m))$ time and $O(m)$ space algorithm to compute all maximal repetitions in a string.

Our algorithm is at least as efficient as the non-RLE algorithms for the general ordered alphabets. Furthermore, when the input string is compressible via RLE, our algorithm can be faster and more space-efficient compared to the non-RLE algorithms. Although our algorithm mimics those for non-RLE strings and is conceptually simple, its correctness is based on new non-trivial observations on the occurrence of specific Lyndon words in run-length encoded strings.

1.2 Organization

The rest of this thesis is organized as follows. In Chapter 2 we give some notations and definitions of several indexing structures. In Chapters 3, 4 and 5, we consider (A) time- and space-efficient construction of string indexing structures: We present the first $O(n)$-time algorithm to construct edge-sorted DAWGs for integer alphabets in Chapter 3. We show a new string indexing structure named the truncated DAWGs, with an application to the length constrained minimal absent words problem in Chapter 4. We then propose a faster and space-saving indexing structure for the left-right maximal generic words problem in Chapter 5. In Chapters 6 and

7, we also consider (B) developing efficient pattern discovery algorithms: We present an online algorithm for enumerating all length-constrained gapped palindromes in an online manner in Chapter 6, and an almost linear time algorithm that enumerates all maximal repetitions in a run length encoded string in Chapter 7.

Chapter 2

Preliminaries

In this chapter, we give notations to be used in this thesis.

2.1 Strings

Let Σ be an alphabet, that is, a nonempty, finite set of characters. A *string* over Σ is a finite sequence of characters from Σ. Let Σ^* denote the set of strings over Σ. The *length* of a string w is the number of characters in w and denoted by $|w|$. Throughout this thesis, we assume the alphabet that Σ is an ordered alphabet. We assume that $\sigma = |\Sigma|$ is a constant or polynomial of input string length n depending on the contexts of problems. The string of length 0 is called the *empty string* and denoted by ε. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any integer $k \geq 0$ and string $x \in \Sigma^*$, $x^0 = \varepsilon$, and $x^k = x^{k-1}x$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$. Strings x, y, and, z are, respectively, said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$. A prefix x, a substring y and a suffix z of w are respectively called a proper prefix, proper substring and proper suffix of w, if $x \neq w$, $y \neq w$ and $z \neq w$. A string x is called a border of w, if it is a proper suffix as well as a prefix of w. The substring of a string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. That is, $w[i..j] = w[i] \cdots w[j]$. For convenience, let $w[i..j] = \varepsilon$ for $i > j$. We use $w[..j]$ and $w[i..]$ as abbreviations of $w[1..j]$ and $w[i..|w|]$. Let $\mathit{Pre}(w)$, $\mathit{Sub}(w)$ and $\mathit{Suf}(w)$ denote the sets of prefixes, substrings, and suffixes of a string w, respectively. For any integer $k \geq 0$ and string $x \in \Sigma^*$, $x^0 = \varepsilon$, and $x^k = x^{k-1}x$. For any string $x \in \Sigma^*$, we define $\mathit{BegPos}(x) = \{i \mid i \in [1, |y| - |x| + 1], y[i..i + |x| - 1] = x\}$, $\mathit{EndPos}(x) = \{i \mid i \in [|x|, |y|], y[i - |x| + 1..i] = x\}$, i.e., the set of beginning and end positions of occurrences of x in y. For any strings u, v, we write $u \equiv_L v$ (resp. $u \equiv_R v$) when $\mathit{BegPos}(u) = \mathit{BegPos}(v)$ (resp. $\mathit{EndPos}(u) = \mathit{EndPos}(v)$). For any string $x \in \Sigma^*$, the equivalence classes with respect to $\equiv_L$ and $\equiv_R$ that x belongs to, are respectively denoted by $[x]_L$ and $[x]_R$. Also, $\overrightarrow{x}$ and $\overleftarrow{x}$ respectively denote the longest elements of $[x]_L$ and $[x]_R$.

For any set S of strings where no two strings are of the same length, let $\mathrm{long}(S) = \arg\max\{|x| \mid x \in S\}$ and $\mathrm{short}(S) = \arg\min\{|x| \mid x \in S\}$.

2.2 Suffix trees and DAWGs

Suffix trees [75] and directed acyclic word graphs (*DAWGs*) [12] are fundamental text data structures. Both of these data structures are based on suffix tries. The *suffix trie* for string y, denoted $\mathit{STrie}(y)$, is a trie representing $\mathit{Sub}(y)$, formally defined as follows.

Definition 1. *$\mathit{STrie}(y)$ for string y is an edge-labeled rooted tree (V_T, E_T) such that*

$$\begin{aligned} V_T &= \{x \mid x \in \mathit{Sub}(y)\} \\ E_T &= \{(x, b, xb) \mid x, xb \in V_T, b \in \Sigma\}. \end{aligned}$$

The second element b of each edge (x, b, xb) is the label of the edge. We also define the set L_T of labeled "reversed" edges called the suffix links *of $\mathit{STrie}(y)$ by*

$$L_T = \{(ax, a, x) \mid x, ax \in \mathit{Sub}(y), a \in \Sigma\}.$$

As can be seen in the above definition, each node v of $\mathit{STrie}(y)$ can be identified with the substring of y that is represented by v. Assuming that string y terminates with a unique character that appears nowhere else in y, for each suffix $y[i..|y|] \in \mathit{Suf}(y)$ there is a unique leaf ℓ_i in $\mathit{STrie}(y)$ such that the suffix $y[i..|y|]$ is spelled out by the path from the root to ℓ_i.

It is well known that $\mathit{STrie}(y)$ requires $O(n^2)$ space. One idea to reduce its space to $O(n)$ is to contract each path consisting only of non-branching edges into a single edge labeled with a non-empty string. Two types of definitions can be made depending on whether or not to leave non branching nodes representing a suffix of the input string. In this theses, a tree that leaves such non-branching nodes is called a suffix tree, and a tree that does not leave nodes is called an Ukkonen tree. Following conventions from [13, 41], suffix tree $\mathit{STree}(y)$ and Ukkonen tree $\mathit{UTree}(y)$ are defined as follows.

Definition 2. *$STree(y)$ for string y is an edge-labeled rooted tree (V_S, E_S) such that*

$$\begin{aligned} V_S &= \{\overrightarrow{x} \mid x \in Sub(y)\} \\ E_S &= \{(x, \beta, x\beta) \mid x, x\beta \in V_S, \beta \in \Sigma^+, b = \beta[1], \overrightarrow{xb} = x\beta\}. \end{aligned}$$

The second element β of each edge $(x, \beta, x\beta)$ is the label of the edge. We also define the set L_S of labeled "reversed" edges called the suffix links *of $STree(y)$ by*

$$L_S = \{(ax, a, x) \mid x, ax \in V_S, a \in \Sigma\},$$

and denote the tree (V_S, L_S) of the suffix links by $SLT(y)$.

Definition 3. *$UTree(y)$ for string y is an edge-labeled rooted tree (V_U, E_U) such that*

$$\begin{aligned} V_U &= \{\overrightarrow{x} \mid x \in Sub(y), \exists a, b \in \Sigma, a \neq b, xa, xb \in Sub(y)\} \\ E_U &= \{(x, \beta, x\beta) \mid x, x\beta \in V_U, \beta \in \Sigma^+, 1 \leq \forall i \leq |\beta|, x\beta \notin V_U\}. \end{aligned}$$

The second element β of each edge $(x, \beta, x\beta)$ is the label of the edge. We also define the set L_S of labeled "reversed" edges called the suffix links *of $UTree(y)$ by*

$$L_U = \{(ax, a, x) \mid x, ax \in V_U, a \in \Sigma\},$$

and denote the tree (V_U, L_U) of the suffix links by $SLT_U(y)$.

By representing each edge label β with a pair of integers (i, j) such that $y[i..j] = \beta$, $STree(y)$ and $UTree(y)$ can be represented with $O(n)$ space. Observe that each internal node of $UTree(y)$ is a branching internal node in $STrie(y)$ and $STree(y\$) = UTree(y\$)$ where \$ is a character that does not appears in y. Note that for any $x \in Sub(y)$ the leaves in the subtree rooted at $\overrightarrow{x}$ correspond to $BegPos(x)$.

An alternative way to reduce the size of $STrie(y)$ to $O(n)$ is to regard $STrie(y)$ as a partial DFA which recognizes $Suf(y)$, and to minimize it. This leads to the directed acyclic word graph $DAWG(y)$ of string y. Following conventions from [13, 41], $DAWG(y)$ is defined as follows.

Definition 4. *$DAWG(y)$ of string y is an edge-labeled DAG (V_D, E_D) such that*

$$\begin{aligned} V_D &= \{[x]_R \mid x \in Sub(y)\} \\ E_D &= \{([x]_R, b, [xb]_R) \mid x, xb \in Sub(y), b \in \Sigma\}. \end{aligned}$$

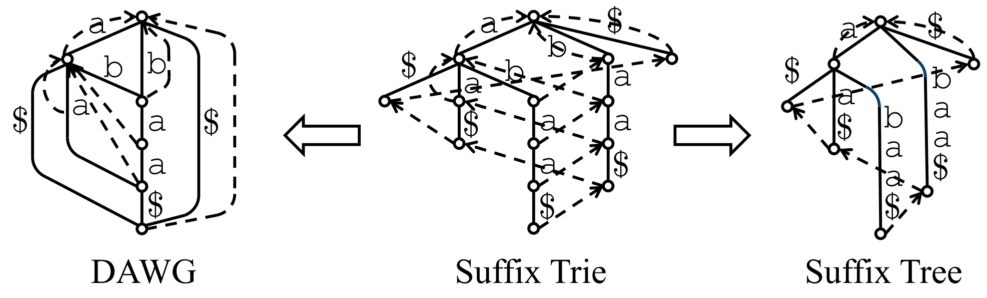


Figure 2.1: $STrie(y)$, $STree(y)$, and $DAWG(y)$ for string $y =$ abaa\$. The solid arcs represent edges, and the broken arcs represent suffix links.

We also define the set L_D of labeled "reversed" edges called the suffix links *of $DAWG(y)$ by*

$$L_D = \{([ax]_R, a, [x]_R) \mid x, ax \in Sub(y), a \in \Sigma, [ax]_R \neq [x]_R\}.$$

See Figure 2.1 for examples of $STrie(y)$, $STree(y)$, and $DAWG(y)$.

Theorem 1 ([12]). *For any string y of length $n > 2$, the number of nodes in $DAWG(y)$ is at most $2n-1$ and the number of edges in $DAWG(y)$ is at most $3n-4$.*

Minimization of $STrie(y)$ to $DAWG(y)$ can be done by merging isomorphic subtrees of $STrie(y)$ which are rooted at nodes connected by a chain of suffix links of $STrie(y)$. Since the substrings represented by these merged nodes end at the same positions in y, each node of $DAWG(y)$ forms an equivalence class $[x]_R$. We will make an extensive use of this property in our $O(n)$-time construction algorithm for $DAWG(y)$ over an integer alphabet.

2.3 Computation Model

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\lceil \log_2 n \rceil$, and hence, standard operations on values representing lengths and positions of strings can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

Chapter 3

Linear-time Computation of DAWGs and Affix Trees for Integer Alphabets

Text indexes are fundamental data structures that allow for efficient processing of string data, and have been extensively studied. Although there are several alternative data structures which can be used as an index, such as suffix trees [75] and suffix arrays [61], in this chapter, we focus on *directed acyclic word graphs* (*DAWGs*) proposed by Blumer et al. [12]. Intuitively, the DAWG of string y, denoted $DAWG(y)$, is an edge-labeled DAG obtained by merging isomorphic subtrees of the trie representing all suffixes of string y, called the suffix trie of y. Hence, $DAWG(y)$ can be seen as an automaton recognizing all suffixes of y. Let n be the length of the input string y. Despite the fact that the number of nodes and edges of the suffix trie can be as large as $O(n^2)$, Blumer et al. [12] proved that, surprisingly, $DAWG(y)$ has at most $2n - 1$ nodes and $3n - 4$ edges for $n > 2$. Crochemore [19] showed that $DAWG(y)$ is the smallest (partial) automaton recognizing all suffixes of y, namely, the sub-tree merging operation which transforms the suffix trie to $DAWG(y)$ indeed minimizes the automaton.

Since $DAWG(y)$ is a DAG, in general, more than one string can be represented by its node. It is known that every string represented by the same node of $DAWG(y)$ has the same set of ending positions in the string y. Due to this property, if z is the longest string represented by a node v of $DAWG(y)$, then any other string represented by the node v is a proper suffix of z. Hence, the *suffix link* of each node of $DAWG(y)$ is well-defined; if ax is the shortest string represented by node v where a is a single character and x is a string, then the suffix link of ax points to the node of $DAWG(y)$ that represents string x.

One of the most intriguing properties of DAWGs is that the suffix links of $DAWG(y)$ for any string y forms the suffix tree [75] of the reversed string of y. Hence, $DAWG(y)$ augmented

Table 3.1: Space requirements and construction times for text indexing structures for input strings of length n over an alphabet of size σ.

	space (in words)	construction time		
		ordered alphabet	integer alphabet	constant alphabet
suffix tries	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
suffix trees	$O(n)$	$O(n \log \sigma)$ [62]	$O(n)$ [28]	$O(n)$ [75]
suffix arrays	$O(n)$	$O(n \log \sigma)$ [62]+[61]	$O(n)$ [28]+[61]	$O(n)$ [75]+[61]
DAWGs	$O(n)$	$O(n \log \sigma)$ [12]	$O(n)$ [this work]	$O(n)$ [12]
CDAWGs	$O(n)$	$O(n \log \sigma)$ [13]	$O(n)$ [66]	$O(n)$ [13]
affix trees	$O(n)$	$O(n \log \sigma)$ [57]	$O(n)$ [this work]	$O(n)$ [57]

with suffix links can be seen as a *bidirectional* text indexing data structure. This line of research was followed by other types of bidirectional text indexing data structures such as *symmetric compact DAWGs* (*SCDAWGs*) [13] and *affix trees* [69, 57]. DAWGs with suffix links also have applications to other kinds of string processing problems which are not always easily solvable by using suffix trees or arrays, such as: finding *minimal absent words* for a given string [23, 70], finding *α-gapped repeats* that occur in a given string [71], finding *maximal-exponent repeats* in a given overlap-free string [4], computing the *Lempel-Ziv 77 factorization* [77] of a given string in an online manner and with compact space [76].

Time complexities for constructing text indexing data structures depend on the underlying alphabet. See Table 3.1. For a given string y of length n over an ordered alphabet of size σ, the suffix tree [62], the suffix array [61], the DAWG, and the *compact DAWGs* (*CDAWGs*) [13] of y can all be constructed in $O(n \log \sigma)$ time. These immediately lead to $O(n)$-time construction algorithms for a constant alphabet.

In this chapter, we are particularly interested in input strings of length n over an *integer alphabet* of polynomial size in n. Farach-Colton et al. [28] proposed the first $O(n)$-time suffix tree construction algorithm for integer alphabets. Since the out-edges of every node of the suffix tree constructed by McCreight's [62] and Farach-Colton et al.'s algorithms are lexicographically sorted, and since sorting is an obvious lower-bound for constructing edge-sorted suffix trees, the above-mentioned suffix-tree construction algorithms are optimal for ordered and integer alphabets, respectively. Since the suffix array of y can be easily obtained in $O(n)$ time from the edge-sorted suffix tree of y, suffix arrays can also be constructed in optimal time. In addition, since the edge-sorted suffix tree of y can easily be constructed in $O(n)$ time from the edge-sorted CDAWG of y, and since the edge-sorted CDAWG of y can be constructed in $O(n)$ time from the edge-sorted DAWG of y [13], sorting is also a lower-bound for constructing edge-sorted DAWGs and edge-sorted CDAWGs. Using the technique of Narisawa et al. [66], edge-sorted CDAWGs can be constructed in optimal $O(n)$ time for integer alphabets. On the other hand, the only known algorithm to construct DAWGs was Blumer et al.'s $O(n \log \sigma)$-time online algorithm [12] for ordered alphabets of size σ, which results in $O(n \log n)$-time DAWG construction for integer alphabets.

In this chapter, we close the gap between the upper and lower bounds for DAWG construction, by proposing the first $O(n)$-time algorithm to construct edge-sorted DAWGs for integer alphabets. Our algorithm also computes the suffix links, and can thus be applied to various kinds of string processing problems. Our algorithm builds $DAWG(y)$ for a given string y by transforming the suffix tree of y to $DAWG(y)$. In other words, our algorithm simulates the minimization of the suffix trie of y to $DAWG(y)$ using only $O(n)$ time and space.

A simple modification to our $O(n)$-time DAWG construction algorithm also leads us to the first $O(n)$-time algorithm to construct affix trees for integer alphabets. We remark that the previous best known affix-tree construction algorithm of Maaß [57] requires $O(n \log n)$ time for integer alphabets.

3.1 Notations

In this chapter, we assume that the input string y of length n is over the integer alphabet $\{1, \ldots, n^C\}$ for some constant C, and that the last character of y is a unique character denoted by \$ that does not occur elsewhere in y.

3.2 Constructing DAWGs in $O(n)$ Time for Integer Alphabet

In this section, we present an optimal $O(n)$-time algorithm to construct $DAWG(y)$ with suffix links L_D for a given string y of length n over an integer alphabet. Our algorithm constructs $DAWG(y)$ with suffix links L_D from $STree(y)$ with suffix links L_S. The following result is known.

Theorem 2 ([28])**.** *Given a string y of length n over an integer alphabet, edge-sorted $STree(y)$ with suffix links L_S can be computed in $O(n)$ time.*

Let $\mathcal{L}$ and $\mathcal{R}$ be, respectively, the sets of longest elements of all equivalence classes on y w.r.t. $\equiv_L$ and $\equiv_R$, namely, $\mathcal{L} = \{\overrightarrow{x} \mid x \in Sub(y)\}$ and $\mathcal{R} = \{\overleftarrow{x} \mid x \in Sub(y)\}$. Let

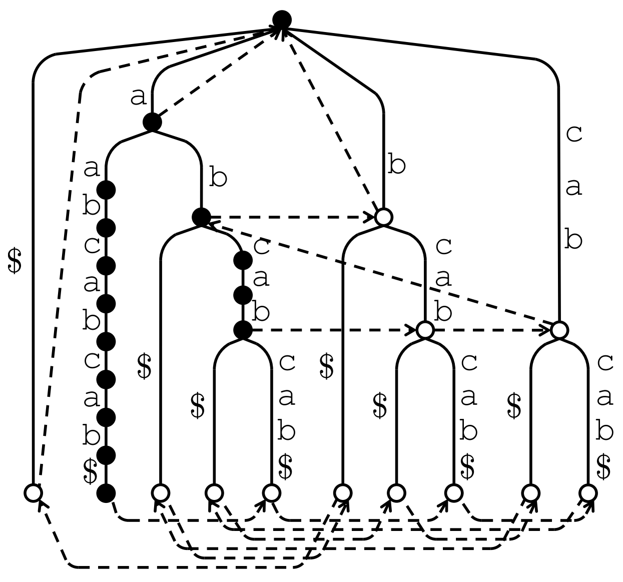


Figure 3.1: An example of $STree'(y)$ with string $y =$ `aabcabcab`$.

$STree'(y) = (V'_S, E'_S)$ be the edge-labeled rooted tree obtained by adding extra nodes for strings in $\mathcal{R}$ to $STree(y)$, namely,

$$\begin{aligned} V'_S &= \{x \mid x \in \mathcal{L} \cup \mathcal{R}\}, \\ E'_S &= \{(x, \beta, x\beta) \mid x, x\beta \in V'_S, \beta \in \Sigma^+, \\ &\qquad 1 \le \forall i < |\beta|, x \cdot \beta[1..i] \notin V'_S\}. \end{aligned}$$

Notice that the size of $STree'(y)$ is $O(n)$, since $|\mathcal{L} \cup \mathcal{R}| \le |V_S| + |V_D| = O(n)$, where V_S and V_D are respectively the sets of nodes of $STree(y)$ and $DAWG(y)$.

A node $x \in V'_S$ of $STree'(y)$ is called *black* iff $x \in \mathcal{R}$. See Figure 3.1 for an example of $STree'(y)$.

Lemma 1. *For any $x \in Sub(y)$, if x is represented by a black node in $STree'(y)$, then every prefix of x is also represented by a black node in $STree'(y)$.*

Proof. Since x is a black node, $x = \overleftarrow{x}$. Assume on the contrary that there is a proper prefix z of x such that z is not represented by a black node. Let $zu = x$ with $u \in \Sigma^+$. Since $z \equiv_R \overleftarrow{z}$, we have $x = zu \equiv_R \overleftarrow{z}u$. On the other hand, since z is not black, we have $|\overleftarrow{z}| > |z|$. However, this contradicts that x is the longest member $\overleftarrow{x}$ of $[x]_R$. Thus, every prefix of x is also represented by a black node.

Lemma 2. *For any string y, let $BT(y)$ be the trie consisting only of the black nodes of $STree'(y)$. Then, every leaf ℓ of $BT(y)$ is a node of the original suffix tree $STree(y)$.*

Proof. Assume on the contrary that some leaf ℓ of $BT(y)$ corresponds to an internal node of $STree'(y)$ that has exactly one child. Since any substring in $\mathcal{L}$ is represented by a node of the original suffix tree $STree(y)$, we have $\ell \in \mathcal{R}$. Since $\ell = \overleftarrow{\ell}$, ℓ is the longest substring of y which has ending positions $EndPos(\ell)$ in y. This implies one of the following situations: (1) occurrences of ℓ in y are immediately preceded by at least two distinct characters $a \neq b$, (2) ℓ occurs as a prefix of y and all the other occurrences of ℓ in y are immediately preceded by a unique character a, or (3) ℓ occurs exactly once in y as its prefix. Let u be the only child of ℓ in $STree'(y)$, and let $\ell z = u$, where $z \in \Sigma^+$. By the definition of ℓ, u is not black. On the other hand, in any of the situations (1)-(3), $u = \ell z$ is the longest substring of y which has ending positions $EndPos(u)$ in y. Hence we have $u = \overleftarrow{u}$ and u must be black, a contradiction. Thus, every leaf ℓ of $BT(y)$ is a node of the original suffix tree $STree(y)$.

Lemma 3 ([66])**.** *For any node $x \in V_S$ of the original suffix tree $STree(y)$, its corresponding node in $STree'(y)$ is black iff (1) x is a leaf of the suffix link tree $SLT(y)$, or (2) x is an internal node of $SLT(y)$ and for any character $a \in \Sigma$ such that $ax \in V_S$, $|BegPos(ax)| \neq |BegPos(x)|$.*

Using Lemma 2 and Lemma 3, we can compute all leaves of $BT(y)$ in $O(n)$ time by a standard traversal on the suffix link tree $SLT(y)$. Then, we can compute all internal black nodes of $BT(y)$ in $O(n)$ time using Lemma 1. Now, by Theorem 2, the next lemma holds:

Lemma 4. *Given a string y of length n over an integer alphabet, edge-sorted $STree'(y)$ can be constructed in $O(n)$ time.*

We construct $DAWG(y)$ with suffix links L_D from $STree'(y)$, as follows. First, we construct a DAG D, which is initially equivalent to the trie $BT(y)$ consisting only of the black nodes of $STree'(y)$. Our algorithm adds edges and suffix links to D, so that the DAG D will finally become $DAWG(y)$. In so doing, we traverse $STree'(y)$ in post-order. For each black node x of $STree'(y)$ visited in the post-order traversal, which is either an internal node or a leaf of the original suffix tree $STree(y)$, we perform the following: Let $p(x)$ be the parent of x in the *original suffix tree $STree(y)$*. It follows from Lemma 1 that every prefix x' of x with $|p(x)| \leq |x'| \leq |x|$ is represented by a black node. For each black node x' in the path from $p(x)$ to x in the DAG D, we compute the in-coming edges to x' and the suffix link of x'.

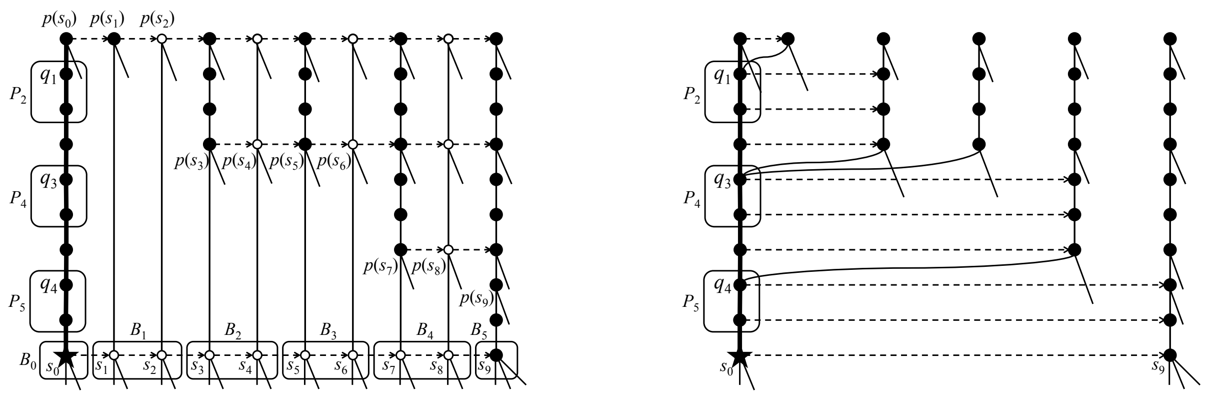


Figure 3.2: (Left): Illustration for a part of $STree'(y)$, where the branching nodes are those that exist also in the original suffix tree $STree(y)$. Suppose we have just visited node $x = s_0$ (marked by a star) in the post-order traversal on $STree'(y)$. Here, $s_0, \ldots, s_9$ are connected by a chain of the suffix links starting from s_0, and s_9 is the first black node after s_0 in the chain. In the corresponding DAG D, we will add in-coming edges to the black nodes in the path from $p(x)$ to x, and will add suffix links from these black nodes in the path. The sequence $s_0, \ldots, s_m$ of nodes in $STree'(y)$ is partitioned into blocks, such that that the parents of the nodes in the same block belong to the same equivalence class w.r.t. $\equiv_R$. (Right): The in-coming edges and the suffix links have been added to the nodes in the path from $p(x)$ to $x = s_0$.

Let $s_0, \ldots, s_m$ be the sequence of nodes connected by a chain of suffix links starting from $s_0 = x$, such that $|BegPos(s_i)| = |BegPos(s_0)|$ for all $0 \leq i \leq m-1$ and $|BegPos(s_m)| > |BegPos(s_0)|$ (see the left diagram of Figure 3.2). In other words, s_m is the first black node after s_0 in the chain of suffix links (this is true by Lemma 3). Since $|s_i| = |s_{i-1}| + 1$ for every $1 \leq i \leq m-1$, $EndPos(s_i) = EndPos(s_0)$. Thus, $s_0, \ldots, s_{m-1}$ form a single equivalence class w.r.t. $\equiv_R$ and are represented by the same node as $x = s_0$ in the DAWG.

For any $0 \leq i \leq m-1$, let $d(s_i) = |s_i| - |p(s_i)|$. Observe that the sequence $d(s_0), \ldots, d(s_m)$ is monotonically non-increasing. We partition the sequence $s_0, \ldots, s_m$ of nodes into blocks so that the parents of all nodes in the same block belong to the same equivalence class w.r.t. $\equiv_R$. Let r be the number of such blocks, and for each $0 \leq k \leq r-1$, let $B_k = s_{i_k}, \ldots, s_{i_{k+1}-1}$ be the kth such block. Note that for each block B_k, $p(s_{i_k})$ is the only black node among the parents $p(s_{i_k}), \ldots, p(s_{i_{k+1}-1})$ of the nodes in B_k, since it is the longest one in its equivalence class $[p(s_{i_k})]_R$. Also, every node in the same block has the same value for function d. Thus, for each block B_k, we add a new edge $(p(s_{i_k}), b_k, q_k)$ to the DAG D, where q_k is the (black) ancestor of x such that $|q_k| = |x| - d(s_{i_k}) + 1$, and b_k is the first character of the label of the edge from $p(s_{i_k})$ to s_{i_k} in $STree'(y)$. Notice that this new edge added to D corresponds to the edges between the nodes in the block B_k and their parents in $STree'(y)$. We also add a suffix link $(p(q_k), a, p(s_{i_k}))$ to D, where $a = s_{i_k - 1}[1]$. See also the right diagram of Figure 3.2.

For each $2 \leq k \leq r-1$, let P_k be the path from q_{k-1} to g_k, where $g_k = p(p(q_k))$ for $2 \leq k \leq r-2$ and $g_{r-1} = x = s_0$. Each P_k is a sub-path of the path from $p(s_0)$ to s_0, and every node in P_k has not been given their suffix link yet. For each node v in P_k, we add the suffix link from v to the ancestor u of s_{i_k} such that $|s_{i_k}| - |u| = |s_0| - |v|$. See also the right diagram of Figure 3.2.

Repeating the above procedure for all black nodes of $STree'(y)$ that are either internal nodes or leaves of the original suffix tree $STree(y)$ in post order, the DAG D finally becomes $DAWG(y)$ with suffix links L_D. We remark however that the edges of $DAWG(y)$ might not be sorted, since the edges that exist in $STree'(y)$ were firstly inserted to the DAG D. Still, we can easily sort all the edges of $DAWG(y)$ in $O(n)$ total time after they are constructed: First, extract all edges of $DAWG(y)$ by a standard traversal on $DAWG(y)$, which takes $O(n)$ time. Next, radix sort them by their labels, which takes $O(n)$ time because we assumed an integer alphabet of polynomial size in n. Finally, re-insert the edges to their respective nodes in the sorted order.

Theorem 3. *Given a string y of length n over an integer alphabet, we can compute edge-sorted $DAWG(y)$ with suffix links L_D in $O(n)$ time and space.*

Proof. The correctness can easily be seen if one recalls that minimizing $STrie(y)$ based on its suffix links produces $DAWG(y)$. The proposed algorithm simulates this minimization using only the subset of the nodes of $STrie(y)$ that exist in $STree'(y)$. The out-edges of each node of $DAWG(y)$ are sorted in lexicographical order as previously described.

We analyze the time complexity of our algorithm. We can compute $STree'(y)$ in $O(n)$ time by Lemma 4. The initial trie for D can easily be computed in $O(n)$ time from $STree'(y)$. Let x be any node visited in the post-order traversal on $STree'(y)$ that is either an internal node or a leaf of the original suffix tree $STree(y)$. The cost of adding the new in-coming edges to the black nodes in the path from $p(x)$ to $x = s_0$ is linear in the number of nodes in the sequence $s_0, \ldots, s_m$ connected by the chain of suffix links starting from $s_0 = x$. Since s_0 and s_m are the only black nodes in the sequence, it follows from Lemma 3 that the chain of suffix links from s_0 to s_m is a non-branching path of the suffix link tree $SLT(y)$. This implies that the suffix links in this chain are used only for node x during the post-order traversal of $STree'(y)$. Since the number of edges in $SLT(y)$ is $O(n)$, the amortized cost of adding each edge to D is constant. Also, the total cost to sort all edges is $O(n)$, as was previously explained. Now let us consider the cost of adding the suffix links from the nodes in each sub-path P_k. For each node v in P_k, the destination node v can be found in constant time by simply climbing up the path from s_{i_k} in the chain of suffix links. Overall, the total time cost to transform the trie for D to $DAWG(y)$ is $O(n)$.

The working space is clearly $O(n)$.

Figure 3.3 shows an example of DAWG construction by our algorithm.

In some applications such as bidirectional pattern searches, it is preferable that the incoming suffix links at each node of $DAWG(y)$ are also sorted in lexicographical order, but our algorithm described above does not sort the suffix links. However, we can sort the suffix links in $O(n)$ time by the same technique applied to the edges of $DAWG(y)$.

3.3 Constructing Affix Trees in $O(n)$ Time for Integer Alphabet

Let y be the input string of length n over an integer alphabet. Recall the sets $\mathcal{L} = \{\overrightarrow{x} \mid x \in \mathit{Sub}(y)\}$ and $\mathcal{R} = \{\overleftarrow{x} \mid x \in \mathit{Sub}(y)\}$ introduced in Section 3.2. For any set $S \subseteq \Sigma^* \times \Sigma^*$ of ordered pairs of strings, let $S[1] = \{x_1 \mid (x_1, x_2) \in S$ for some $x_2 \in \Sigma^*\}$ and $S[2] = \{x_2 \mid (x_1, x_2) \in S$ for some $x_1 \in \Sigma^*\}$. For any string x, let x^R denote the reversed string of x.

The affix tree [69] of string y, denoted $\mathit{ATree}(y)$, is a *bidirectional* text indexing structure defined as follows:

Definition 5. *$\mathit{ATree}(y)$ for string y is an edge-labeled DAG $(V_A, E_A) = (V_A, E_A^F \cup E_A^B)$ which has two mutually distinct sets E_A^F, E_A^B of edges such that*

$$\begin{aligned}
V_A &= \{(x, x^R) \mid x \in \mathcal{L} \cup \mathcal{R}\}, \\
E_A^F &= \{((x, x^R), \beta, (x\beta, \beta^R x^R)) \mid x, x\beta \in V_A[1], \\
&\qquad \beta \in \Sigma^+, 1 \le \forall i < |\beta|, x \cdot \beta[1..i] \notin V_A[1]\}, \\
E_A^B &= \{((x, x^R), \alpha^R, (\alpha x, x^R \alpha^R)) \mid x^R, x^R\alpha^R \in V_A[2], \\
&\qquad \alpha \in \Sigma^+, 1 \le \forall i < |\alpha|, x^R \cdot \alpha^R[1..i] \notin V_A[2]\}.
\end{aligned}$$

E_A^F is the set of forward edges labeled by substrings of y, while E_A^B is the set of backward edges labeled by substrings of y^R.

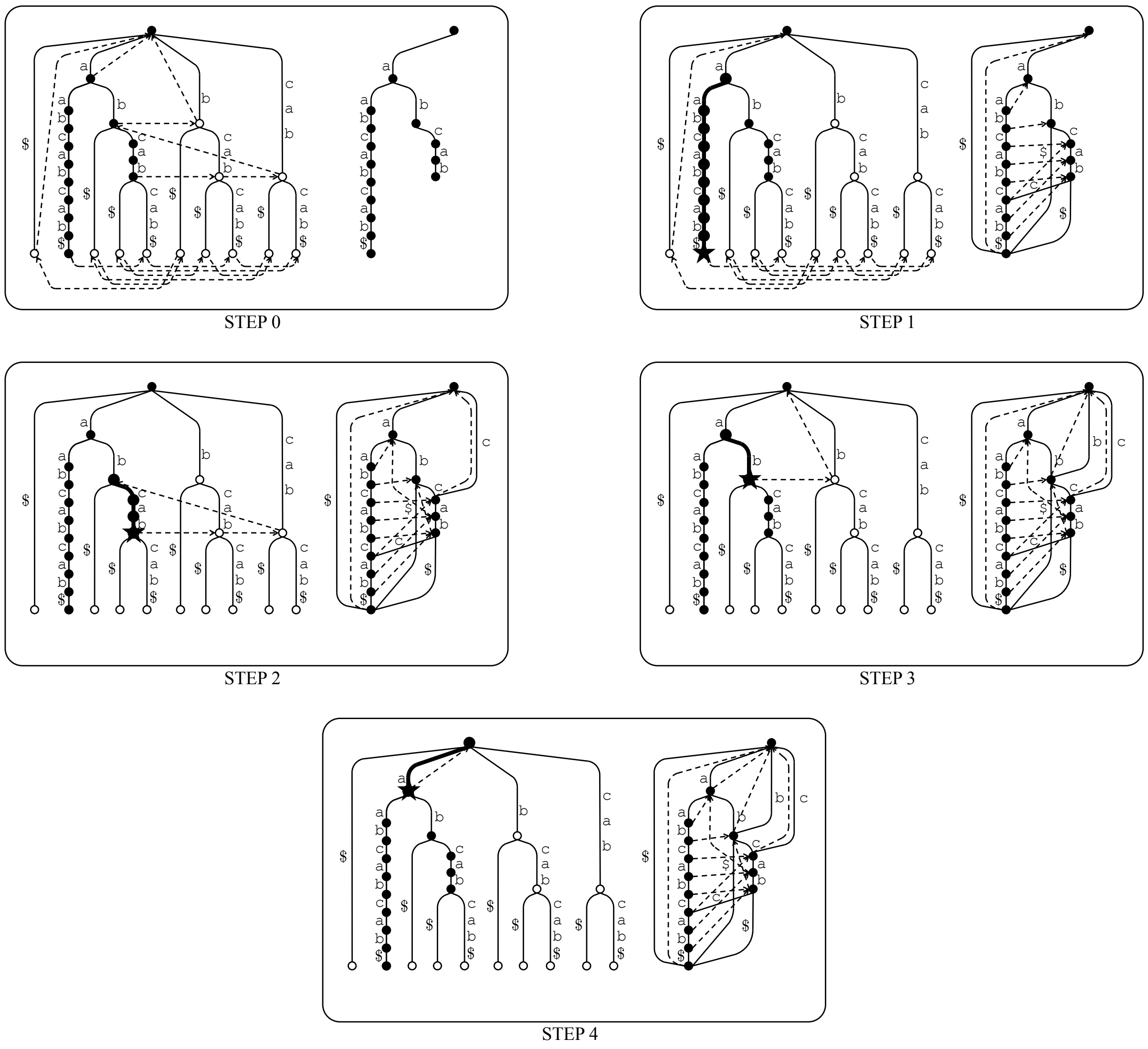


Figure 3.3: Snapshots during the construction of $DAWG(y)$ for $y = \texttt{aabcabcab\$}$. Step 0: (Left): $STree'(y)$ with suffix links L_S and (Right): the initial trie for D. We traverse $STree'(y)$ in post order. Step 1: We arrived at black leaf node $x_1 = \texttt{aabcabcab\$}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_1) = \texttt{a}$ and x_1 (indicated by thick black lines). To the right is the resulting DAG D for this step. Step 2: We arrived at black branching node $x_2 = \texttt{abcab}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_2) = \texttt{ab}$ and x_2 (indicated by thick black lines). To the right is the resulting DAG D for this step. Step 3: We arrived at black branching node $x_3 = \texttt{ab}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_3) = \texttt{a}$ and x_3 (indicated by thick black lines). To the right is the resulting DAG D for this step. Step 4: We arrived at black branching node $x_4 = \texttt{a}$ (indicated by a star). We determine the in-coming edges and suffix links for the black nodes in the path from $p(x_4) = \varepsilon$ and x_4 (indicated by thick black lines). To the right is the resulting DAG D for this step. Since all branching and leaf black nodes have been processed, the final DAG D is $DAWG(y)$ with suffix links.

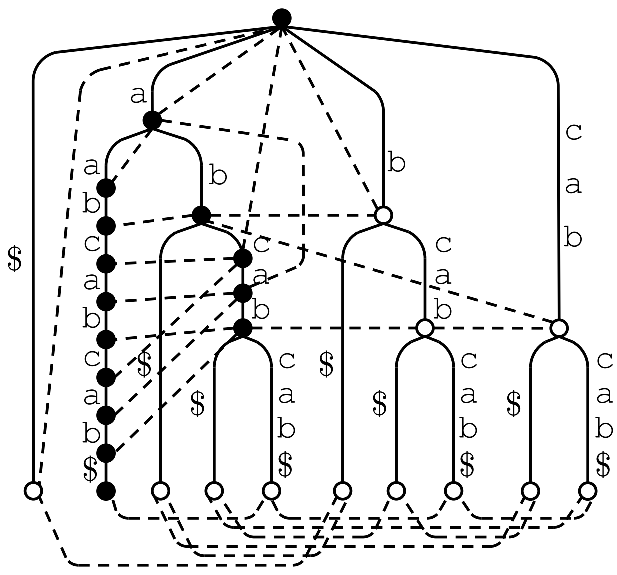


Figure 3.4: An example of $ATree(y)$ with string $y =$ aabcabcab\$. The solid arcs represent the forward edges in E^F_A, while the broken arcs represent the backward edges in E^B_A. For simplicity, the labels of backward edges are omitted.

Theorem 4. *Given a string y of length n over an integer alphabet, we can compute edge-sorted $ATree(y)$ in $O(n)$ time and space.*

Proof. Clearly, there is a one-to-one correspondence between each node $(x, x^R) \in V_A$ of $ATree(y) = (V_A, E^F_A \cup E^B_A)$ and each node $x \in V'_S$ of $STree'(y) = (V'_S, E'_S)$ of Section 3.2 (see also Figure 3.1 and Figure 3.4). Moreover, there is a one-to-one correspondence between each forward edge $(x, \beta, x\beta) \in E^F_A$ of $ATree(y)$ and each edge $(x, \beta, x\beta) \in E'_S$ of $STree'(y)$. Hence, what remains is to construct the backward edges in E^B_A for $ATree(y)$. A straightforward modification to our DAWG construction algorithm of Section 3.2 can construct the backward edges of $ATree(y)$; instead of working on the DAG D, we directly add the suffix links to the black nodes of $STree'(y)$ whose suffix links are not defined yet (namely, those that are neither branching nodes nor leaves of the suffix link tree $SLT(y)$). Since the suffix links are reversed edges, by reversing them we obtain the backward edges of $ATree(y)$. The labels of the backward edges can be easily computed in $O(n)$ time by storing in each node the length of the string it represents. Finally, we can sort the forward and backward edges in lexicographical order in overall $O(n)$ time, using the same idea as in Section 3.2.

Chapter 4

Truncated DAWGs: Efficient Data Structures for Processing Short Substrings

Text indexing structures are very important to perform string processing efficiently such as pattern matching and longest common extension queries. Several indexing structures such as suffix trees [75], suffix arrays [61] and directed acyclic word graphs (DAWGs) [12, 19] can represent all suffixes of a string in linear space with respect to the length of the string. However, it is not efficient to store all suffixes of the string in some applications, for example when we only want to find short keywords in very long texts. Na et al. [65] proposed *k-truncated suffix trees* which are the pruned version of suffix trees that require less space than the suffix trees in practice. The k-truncated suffix tree of a string y is a compressed trie that represents substrings of y whose length is less than or equal to k. They also show an application of truncated suffix trees for LZ77 [77] that compresses using a sliding window of a fixed size [65]. Later, Tanimura et al. [72] showed that the k-truncated suffix tree of a string y can be represented in $O(\min\{n, kZ\})$ space, where n is the length of y and Z is the size of LZ77 factorization of y.

In this chapter, we focus on *directed acyclic word graphs* (*DAWGs*) [12]. The DAWG of a string y, denoted by $DAWG(y)$, is an edge-labeled directed acyclic graph obtained by merging isomorphic subtrees of the suffix trie of y. It is known that each node in $DAWG(y)$ represents substrings of y that have the same set of ending positions. On the other hand, $DAWG(y)$ also can be seen as the smallest automaton recognizing all suffixes of y. We can make the smallest automaton recognizing all substrings of length k or less, by minimizing the truncated suffix trie of y, which represents substrings of y whose length is less than or equal to k (see Figure 4.1). However, it is difficult to construct such automaton and sometimes its size does not become small, for example, when all characters in y are different from one another (see Figure 4.2).

In this chapter, we propose a new data structure called *k-truncated DAWG*, which is the DAWG with some of its nodes and edges deleted. The k-truncated DAWG of y, denoted by $k\text{-}TDAWG(y)$, is a subgraph of $DAWG(y)$ where a node in $DAWG(y)$ is also a node in $k\text{-}TDAWG(y)$ if and only if the length of the shortest string represented by the node in $DAWG(y)$ is k or less. We show that the $k\text{-}TDAWG(y)$ can be stored in $O(\min\{n, k\gamma\})$ space, where n is the length of y and γ is the size of one of the smallest k-attractors of y [42]. We also present an online algorithm that constructs $k\text{-}TDAWG(y)$ in $O(n \log \sigma)$ time and $O(\min\{n, k\gamma\})$ space, where σ is the alphabet size. We modify the online DAWG construction algorithm by Blumer et al. [12]. by adding node and edge deletion operations to the algorithm and show that these deletion operations can be performed safely while maintaining $O(\min\{n, k\gamma\})$ working space.

For a string y, it is known that the suffix links of the $DAWG(y)$ coincide with the edges of the suffix tree of y^R [25], where y^R is the reverse string of y. We show that this property also holds between the k-truncated DAWG of y and the k-truncated suffix tree of y^R. Thus we can simulate the k-truncated suffix tree of y^R on $k\text{-}TDAWG(y)$ by using the reversed suffix links of $k\text{-}TDAWG(y)$ with label addition. Moreover, the truncated DAWG of y contains secondary edges, which are not present in truncated suffix tree of y^R.

As an application of $k\text{-}TDAWG(y)$, we present an algorithm to compute the set $MAW_k(y)$ of all minimal absent words of y whose length is smaller than or equal to k by using $k\text{-}TDAWG(y)$. A string x is said to be a minimal absent word of y if x does *not* occur in y and all proper substrings of x occur in y. Minimal absent words have some applications such as to build phylogeny [14] and pattern matching [20]. Let $MAW(y)$ be the set of minimal absent words of y. Fujishige et al. [32] proposed an algorithm to compute $MAW(y)$ by using $DAWG(y)$ in $O(n + |MAW(y)|)$ time. This problem cannot be solved using the suffix tree of y and its suffix links in the same time and space complexity. In this chapter, we show that $MAW_k(y) = \{x \mid x \in MAW(y), |x| \leq k\}$ can be computed by using k-truncated DAWG in $O(\min\{n, k\gamma\} + |MAW_k(y)|)$ time. Similar to $MAW(y)$, $MAW_k(y)$ cannot be computed using the truncated suffix tree of y with its suffix links in the same time and space complexity.

We also consider k-truncated DAWGs for multiple strings. For a set of strings S, the k-truncated DAWG of S, denoted by $k\text{-}TDAWG(S)$, is a subgraph of the $DAWG(S)$ (see [13]), where a node in $DAWG(S)$ is also a node in $k\text{-}TDAWG(S)$ if and only if the length of the shortest string represented by the node is k or less. We show that the size of $k\text{-}TDAWG(S)$

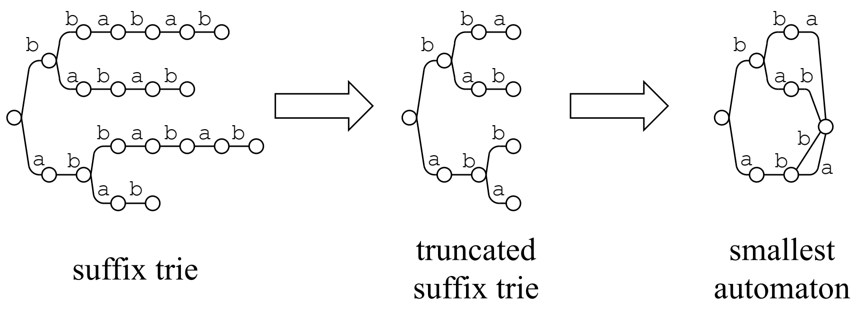


Figure 4.1: The suffix trie of y = abbabab, the truncated suffix trie of y and the smallest automaton which represents all substrings of y of length 3 or less.

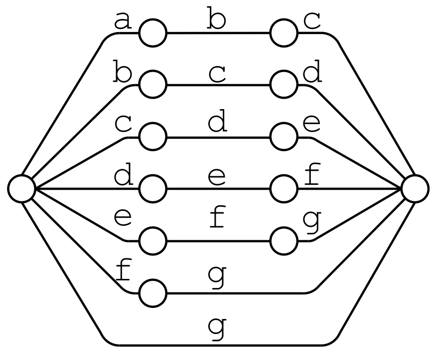


Figure 4.2: The smallest automaton which represents all substrings of y = abcdefg of length 3 or less.

is $O(\min\{N, k(\lambda + m)\})$, where N is the total length of all patterns in S, m is the number of patterns in S, and λ is the size of one of the smallest k-attractors of S. We also show that k-$TDAWG(S)$ can be constructed and used in the same manner as that for the k-truncated DAWG of a single string.

Last, we check the size of k-truncated DAWGs compared to the size of DAWGs by experiments. The experimental results show that the size of k-truncated DAWGs is much smaller than DAWGs when k is small and the string is repetitive. Moreover, the construction time of k-truncated DAWGs is also faster than that of DAWGs when the string is repetitive.

A preliminary version of this article was presented in [31]. In this chapter, we show a tighter upper bound of the size of k-truncated DAWGs and define k-truncated DAWGs for multiple strings.

4.1 Definitions

4.1.1 LZ77 factorization

The *Lempel-Ziv 77 factorization* (*LZ77 factorization*) with self-references [77] of a string y is a sequence $f_1 f_2 \cdots f_Z = y$ that satisfies the following conditions:

- $f_1 = y[1]$,
- $f_i = y[|f_1 \cdots f_{i-1}| + 1]$ if $y[|f_1 \cdots f_{i-1}| + 1]$ does not occur in $f_1 \cdots f_{i-1}$,
- otherwise, f_i is the longest prefix of $y[|f_1 \cdots f_{i-1}|+1..|y|]$ such that f_i begins at a position inside $y[1..|f_1 \cdots f_{i-1}|]$.

In this Chapter, Z denotes the size of LZ77 factorization of y. For example, LZ77 factorization of $y = \texttt{ababbbabbba}$ is $f_1 = \texttt{a}, f_2 = \texttt{b}, f_3 = \texttt{ab}, f_4 = \texttt{bb}, f_5 = \texttt{abbba}$ and this factorization size is 5.

4.1.2 String attractors

The *string attractors* and *k-attractors* [42] of a string y is the set of positions that is defined as follow.

Definition 6 ([42]). *A string attractor Γ of a string y is a set of γ positions of y such that any substring $y[i..j]$ has an occurrence $y[i'..j'] = y[i..j]$ with $\tau \in [i', j']$ for some $\tau \in \Gamma$.*

Definition 7 ([42]). *A k-attractor Γ of a string y is a set of γ positions of y such that any substring $y[i..j]$ with $i \leq j < i + k$ has an occurrence $y[i'..j'] = y[i..j]$ with $\tau \in [i', j']$ for some $\tau \in \Gamma$. When $k = n$, a k-attractor is a string attractor of y.*

The following lemma holds for the size of k-attractors and substrings.

Lemma 5. *$|Sub_k(y)|$ is in $O(\min\{n, k\gamma\})$, where n is the size of y and γ is the size of one of the smallest k-attractors of y.*

Proof. From Lemma 1 in [68], the number of distinct substrings of y with length k is in $O(k\gamma)$. The number of suffixes of y with length less than k is $k-1$. Thus $|Sub_k(y)|$ is in $O(\min\{n, k\gamma\})$.

4.1.3 Truncated suffix trees

Na et al. [65] proposed *k-truncated suffix trees* which can be obtained by pruning some branches of suffix trees. The k-truncated suffix tree of y is a compressed trie that represents substrings of y whose length is less or equal to k. Formally, the k-truncated suffix tree $k\text{-}\mathit{TSTree}(y)$ for string y is defined as follows.

Definition 8. *The k-truncated suffix tree $k\text{-}\mathit{TSTree}(y)$ for a string y is an edge-labeled rooted tree $(V_{k\text{-}TS}, E_{k\text{-}TS})$ such that*

$$\begin{aligned} V_{k\text{-}TS} &= \{[x]_L \mid x \in \mathit{Sub}_k(y)\} \\ E_{k\text{-}TS} &= \{([x]_L, b\beta, [xb]_L) \mid [x]_L, [xb]_L \in V_{k\text{-}TS}, [x]_L \neq [xb]_L, \overrightarrow{xb}[1..\min\{k, |\overrightarrow{xb}|\}] = xb\beta\} \end{aligned}$$

Figure 4.3 shows an example of the k-truncated suffix tree. The truncated suffix trees is useful for LZ77 that compresses using a sliding window of a fixed size [65]. Since basic operations on suffix trees can be simulated against strings of length k or less on truncated suffix trees, truncated suffix trees can be used as suffix trees for some algorithms such as data compression [65] and pattern matching [72]. The following lemma holds for the size of $k\text{-}\mathit{TSTree}(y)$.

Lemma 6 ([2]). *The k-truncated suffix tree of y can be computed in $O(n \log \sigma)$ time and represented in $O(|\mathit{Sub}_k(y)|)$ space.*

From Lemma 5 and 6, the following lemma holds.

Lemma 7. *The size of the k-truncated suffix tree of y is in $O(\min\{n, k\gamma\})$*

4.2 k-truncated DAWGs

In this section, we present a new data structure called *truncated DAWGs*, which is data structures that can be obtained by deleting some nodes and edges of the DAWGs. We also show some properties of truncated DAWGs. First, we define k-truncated DAWGs as follows.

Definition 9. *The k-truncated directed acyclic word graph $k\text{-}\mathit{TDAWG}(y)$ for a string y is a directed graph $(V_{k\text{-}TD}, E_{k\text{-}TD})$ such that*

$$\begin{aligned} V_{k\text{-}TD} &= \{[x]_R \mid x \in \mathit{Sub}_k(y)\} \\ E_{k\text{-}TD} &= \{([x]_R, b, [xb]_R) \mid x, xb \in \mathit{Sub}_k(y), b \in \Sigma\}. \end{aligned}$$

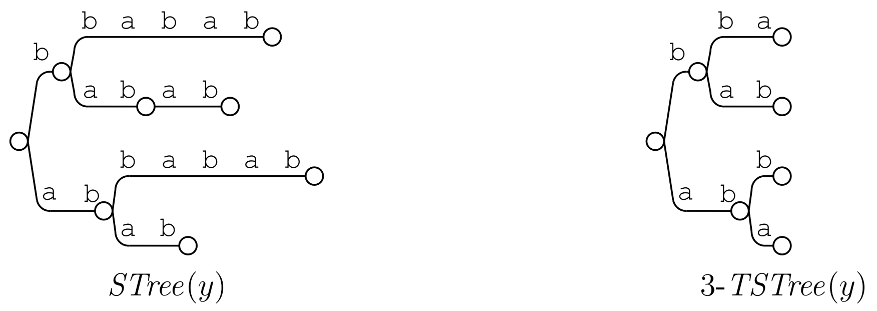


Figure 4.3: The suffix tree and 3-truncated suffix tree of $y = \texttt{abbabab}$.

For any node $v \in V_{k\text{-}TD}$ of $k\text{-}TDAWG(y)$ and character $b \in \Sigma$, we write $\delta_{TD}(v, b) = u$ if $(v, b, u) \in E_{k\text{-}TD}$ for some $u \in V_{k\text{-}TD}$, and $\delta_{TD}(v, b) = nil$ otherwise.

We also define the set $L_{k\text{-}TD}$ of labeled "reversed" edges called the *suffix links* of $k\text{-}TDAWG(y)$ by

$$L_{k\text{-}TD} = \{([ax]_R, a, [x]_R) \mid x, ax \in Sub(y), [ax]_R \in V_{k\text{-}TD}, a \in \Sigma, [ax]_R \neq [x]_R\}.$$

For any suffix link $(u, a, v) \in L_{k\text{-}TD}$ of $k\text{-}TDAWG(y)$, we write $sl_{TD}(u) = v$. There is exactly one suffix link coming out from each node $u \in V_{k\text{-}TD} \setminus \{[\epsilon]_R\}$ of $k\text{-}TDAWG(y)$, so the character a is unique for each node u.

By the definition, clearly $V_{k\text{-}TD} \subseteq V_D, E_{k\text{-}TD} \subseteq E_D$, and $L_{k\text{-}TD} \subseteq L_D$ hold. See Figure 4.4 for examples of $DAWG(y)$ and $3\text{-}TDAWG(y)$. Because $k\text{-}TDAWG(y)$ is a subgraph of $DAWG(y)$, the size of $k\text{-}TDAWG(y)$ is smaller than or equal to the size of $DAWG(y)$ (see Figure 4.5). From the definition of truncated DAWGs, $k\text{-}TDAWG(y)$ can simulate δ_D and sl_D on strings of length k or less similar to $DAWG(y)$.

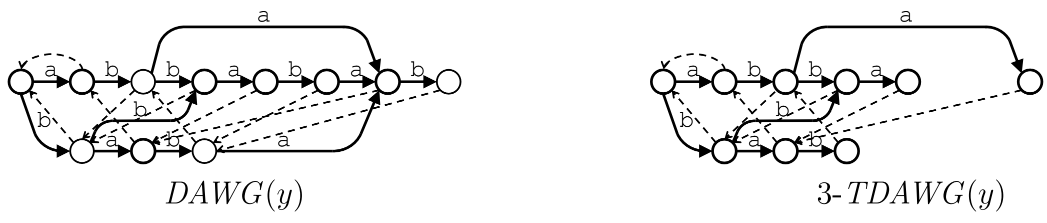


Figure 4.4: The DAWG and 3-truncated DAWG for $y = \texttt{abbabab}$. The solid arcs represent edges and the broken arcs represent suffix links.

For a string y, it is known that the number of nodes of $DAWG(y)$ coincides with the number of nodes of $STree(y^R)$ [25]. From the definition, this property also holds between $k\text{-}TDAWG(y)$ and $k\text{-}TSTree(y^R)$.

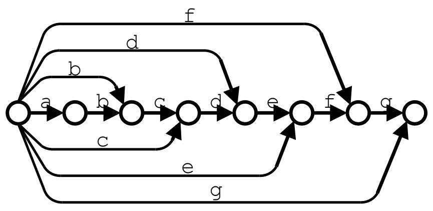


Figure 4.5: *3-TDAWG*(y) = *DAWG*(y) of $y = \texttt{abcdefg}$.

Lemma 8. *For any string y, k-TSTree(y^R) and a tree $(V_{k\text{-}TD}, L_{k\text{-}TD})$ composed of suffix links and nodes of k-TDAWG(y) are isomorphic.*

Proof. For convenience, let $BegPos^R(x)$ be the set of beginning positions of x in y^R, $s \equiv^R_L t$ if and only if $BegPos^R(s) = BegPos^R(t)$ and $[x]^R_L$ be equivalence classes with respect to $\equiv^R_L$. $BegPos^R(x^R)$ can be defined as $\{n - i + 1 \mid i \in EndPos(x)\}$. Since $EndPos(s) = EndPos(t)$ if and only if $BegPos^R(s^R) = BegPos^R(t^R)$, $s \equiv_R t$ if and only if $s^R \equiv^R_L t^R$. Because $[x]_R$ to $[x^R]^R_L$ mapping is a one-to-one correspondence, the nodes of k-*TDAWG*(y) and nodes of k-*TSTree*(y^R) also have one-to-one correspondence. Moreover, from definition of the truncated suffix trees and the truncated DAWGs, the edges $([x]_L, b\beta, [xb]_L)$ of k-*TSTree*(y^R) and the suffix links $([bx^R]^R_L, b, [x^R]^R_L)$ of truncated DAWG have one-to-one correspondance. Therefore, the tree composed of suffix links of k-truncated DAWG and suffix tree are isomorphic.

Theorem 5. *Let $ML_{k\text{-}TD}$ be label-modified suffix links of k-TDAWG(y) which can be obtained by modifying a label a of each suffix link $([ax]_R, a, [x]_R)$ of k-TDAWG(y) to αa, such that $\alpha \in \Sigma^*$ satisfies $|\alpha a x| \le k$ and $\alpha a x = \overleftarrow{ax}$ or $|\alpha a x| > k$ and $\overleftarrow{\alpha a x} = \overleftarrow{ax}$ and $|\alpha a x| = k$. Then the tree $(V_{k\text{-}TD}, ML_{k\text{-}TD})$ is k-TSTree(y^R).*

By using the Lemma 8, we can show that the following theorem holds for the space complexity of k-truncated DAWGs.

Theorem 6. *Given a string y of length n, k-TDAWG(y) can be stored in $O(\min\{n, k\gamma\})$ space, where γ is the size of one of the smallest k-attractors of y.*

Proof. First, we prove $|V_{k\text{-}TD}| \in O(\min\{n, k\gamma\})$. From Lemma 8, the number of nodes of k-*TDAWG*(y) is the same as the number of nodes of k-*TSTree*(y^R). Since $|Sub_k(y^R)| = |Sub_k(y)| \in O(\min\{n, k\gamma\})$ and the number of nodes of k-*TSTree*(y^R) is $O(|Sub_k(y^R)|)$ by Lemma 7, thus $|V_{k\text{-}TD}| \in O(\min\{n, k\gamma\})$. Next, we prove $|E_{k\text{-}TD}| \in O(\min\{n, k\gamma\})$. Let

$l(v)$ be $\arg\min_x\{|x| \mid x \in v\}$ for each node $v \in V_{k\text{-}TD}$. Consider a spanning tree T on the k-$TDAWG$ consisting of the shortest path from the root to each node, the number of edges in T is obviously $O(\min\{n, k\gamma\})$. Let E be the set of edges of k-$TDAWG(y)$ not included in T. For $\lambda = ([x]_R, b, [xb]_R) \in E$, consider a function f, $f(\lambda) = \text{short}([x]_R) \cdot s$, where $|\text{short}([x]_R) \cdot s| = k, s[1] = b$ and $\text{short}([x]_R) \cdot s \in Sub(y)$. Since, f is injective function from E to k-mers of y, $|E| \in O(k\gamma)$. Moreover, $E_{k\text{-}TD} \in O(n)$ because $E_{k\text{-}TD} \subset E_D$. Therefore, $|E_{k\text{-}TD}| \in O(\min\{n, k\gamma\})$.

Theorem 7. *For two strings $x, y \in \Sigma^*$, k-$TSTree(x) = k$-$TSTree(y)$ and k-$TDAWG(x^R) = k$-$TDAWG(y^R)$ if $Sub_k(x) = Sub_k(y)$.*

Proof. From the definition of k-truncated suffix trees, k-$TSTree(x)$ is the Patricia tree of $Sub_k(x)$, thus k-$TSTree(x) = k$-$TSTree(y)$ if $Sub_k(x) = Sub_k(y)$. Since the node set of k-$TSTree(x)$ coincides with k-$TSTree(y)$, the node set of k-$TDAWG(x^R)$ matches its of k-$TDAWG(y^R)$. From the definition of edge set of truncated DAWGs, k-$TDAWG(x^R)$ coincides with k-$TDAWG(y^R)$.

4.3 Construction of Truncated DAWGs

In this section, we present an online construction algorithm of k-truncated DAWGs. As previously mentioned, since k-$TDAWG(y)$ is a subgraph of $DAWG(y)$, k-$TDAWG(y)$ can be constructed in $O(n \log \sigma)$ time and $O(n)$ working space by traversing all edges and vertices of $DAWG(y)$ and deleting unnecessary ones. However, the working space of this procedure is not optimal, because we need to construct $DAWG(y)$. Therefore, we propose an optimal working space algorithm that can construct k-$TDAWG(y)$ in $O(n \log \sigma)$ time and $O(\min\{n, k\gamma\})$ working space. We modify the online DAWG construction algorithm by Blumer et al. [12] by adding deletion operations to the algorithm.

The main idea of our algorithm is that the algorithm deletes unnecessary nodes and edges while creating new nodes and edges similarly to $DAWG(y)$ construction algorithm. In order to show that k-$TDAWG(y)$ can be constructed similarly to $DAWG(y)$, first we show the following lemma.

Lemma 9. *Let v be a node of both $DAWG(y[1..i])$ and $DAWG(y[1..i+1])$. If v does not exist in k-$TDAWG(y[1..i])$, v also does not exist in k-$TDAWG(y[1..i+1])$.*

Algorithm 1: An $O(n \log \sigma)$-time construction algorithm of $k\text{-}TDAWG(y)$

```
Input: string $y$ of length $n$.
Output: $k\text{-}TDAWG(y) = (V_{k\text{-}TD}, E_{k\text{-}TD})$ and $L_{k\text{-}TD}$
1  $V, E$ and $L$ are empty;
2  make new node $v$;
3  $V \leftarrow V \cup \{v\}$;
4  for $i = 1$ to $n$ do
5      $u \leftarrow v$;
6      make a new node $v$;
7      $V \leftarrow V \cup \{v\}$;
8      while $\delta_{TD}(u, y[i]) = nil$ and $sl_{TD}(u) \neq nil$ and $|\text{short}(u)| < k$ do
9          $E \leftarrow E \cup \{(u, y[i], v)\}$;
10         $u \leftarrow sl_{TD}(u)$;
11     if $(u, y[i], \delta_{TD}(u, y[i]))$ is a secondary edge then
12         if $|\text{short}(u)| = k$ then
13             $E \leftarrow E \backslash \{(u, a, \delta_{TD}(u, a)) \mid \forall a \in \Sigma\}$;
14         $split(u, y[i])$;
15     if there is no in-degree edge of $v$ then
16         $V \leftarrow V \backslash v$;
17         $v \leftarrow \delta_{TD}(u, y[i])$;
18     $L \leftarrow L \cup \{(v, \delta_{TD}(u, y[i]))\}$;
19 Output $(V, E)$ and $L$;
```

Proof. Let, $[x]_R^i$ be the equivalence class represented by v in $DAWG(y[1..i])$ and $[x]_R^{i+1}$ be the equivalence class represented by v in $DAWG(y[1..i+1])$. Assume that v is not in $k\text{-}TDAWG(y[1..i])$ and in $k\text{-}TDAWG(y[1..i+1])$. From the assumption, there is a string w such that $w \notin [x]_R^i$, $w \in [x]_R^{i+1}$, and $|w| \leq k$. However, $[x]_R^{i+1} \subseteq [x]_R^i$ holds from the definition, which is a contradiction.

By Lemma 9, we can safely delete nodes which are in $k\text{-}TDAWG(y[1..i-1])$ but not in $k\text{-}TDAWG(y[1..i])$ and do not need to consider the nodes that have been deleted when constructing $k\text{-}TDAWG(y)$ in an online manner. Thus, we can construct $k\text{-}TDAWG(y[1..i+1])$ from $k\text{-}TDAWG(y[1..i])$ online in a similar way to the DAWG construction algorithm in [12]. Algorithm 1 shows a pseudo-code of the proposed algorithm, provided that the function *split* is shown in Algorithm 3. For strings s and t ($|s| < |t|$) which holds $[s]_R^{i-1} = [t]_R^{i-1}$ and $[s \cdot y[i]]_R^i \neq [t \cdot y[i]]_R^i$, the function *split* compute $[sy[i]]_R^i$ and $[ty[i]]_R^i$ by spliting the node $[s]_R^{i-1} = \delta_D(u, y[i])$. Figure 4.6 shows a snapshot during the construction of $3\text{-}TDAWG(y)$ for $y = \texttt{abbabab}$.

Algorithm 2: An $O(n \log \sigma)$-time construction algorithm of $DAWG(y)$[12]

```
   Input: string $y$ of length $n$.
   Output: $DAWG(y) = (V_D, E_D)$ and $L_D$
1  $V, E$ and $L$ are empty;
2  make new node $v$;
3  $V \leftarrow V \cup \{v\}$;
4  for $i = 1$ to $n$ do
5      $u \leftarrow v$;
6      make a new node $v$;
7      $V \leftarrow V \cup \{v\}$;
8      while $\delta_D(u, y[i]) = nil$ and $sl_D(u) \neq nil$ do
9          $E \leftarrow E \cup \{(u, y[i], v)\}$;
10         $u \leftarrow sl_D(u)$;
11     if $(u, y[i], \delta_D(u, y[i]))$ is a secondary edge then
12         $split(u, y[i])$;
13     $L \leftarrow L \cup \{(v, \delta_D(u, y[i]))\}$;
14 Output $(V, E)$ and $L$;
```

Algorithm 1 satisfies the following theorem.

Theorem 8. *Given a string y of length n over an ordered alphabet and a natural number k, Algorithm 1 computes k-$TDAWG(y)$ in $O(n \log \sigma)$ time and $O(\min\{n, k\gamma\})$ working space in an online manner, where γ is the size of one of the smallest string attractors of y.*

Proof. Algorithm 1 shows the construction algorithm of k-$TDAWG(y)$ and Algorithm 2 shows the construction algorithm of $DAWG(y)$. The differences between these algorithms are operations of deleting nodes and edges in lines 12–13 and lines 15–17 of Algorithm 1. First, we show the correctness of our algorithm. By the definition of truncated DAWGs, it is clear that truncated DAWGs moves in the same manner as DAWGs for $u \in V_{k\text{-}TD}$. For each step, because algorithm runs on the nodes u which corresponded to $y[i-k..i-1]$ and connected nodes by suffix links, any nodes v such that $|\text{short}(v)| > k$ is never visited (see Lemma 9). Therefore the nodes and edges which corresponded only to strings whose length is greater than k can be deleted immediately. Thus we can construct truncated DAWGs in an online manner by adding delete operation of nodes and edges in each step.

Next, we prove the working space in $O(\min\{n, k\gamma\})$. The nodes and edges are deleted only in lines 13 and 16. These deleted nodes and edges are made in line 6 or line 14, their size is obviously not over the size of k-truncated DAWG of $y[1..i-1]$. Thus working space complexity is $O(k\gamma)$.

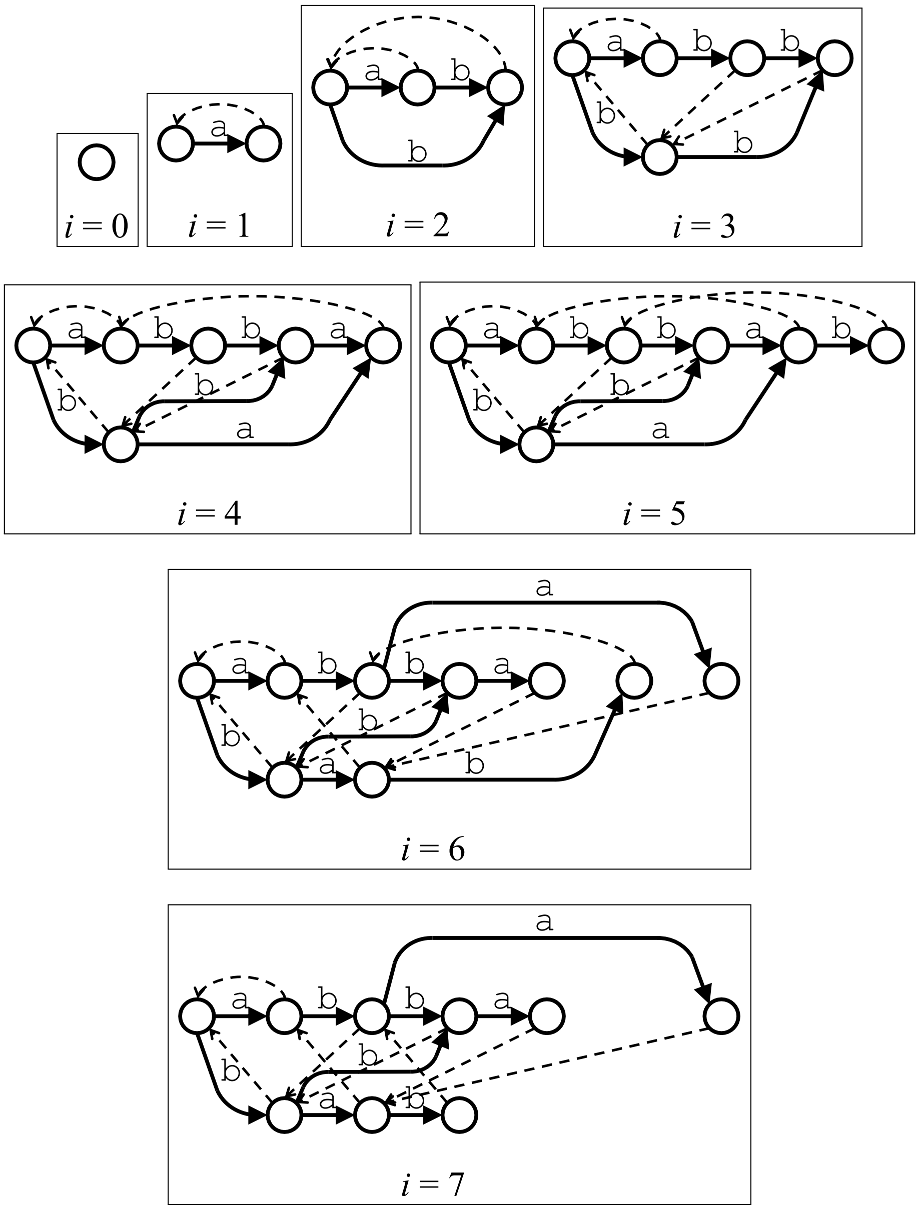


Figure 4.6: Snapshots during the construction of $3\text{-}TDAWG(y)$ for $y =$ `abbabab` on Algorithm 1.

4.4 Applications of Truncated DAWGs for Minimal Absent Words

In this section, we show an algorithm to compute all minimal absent words of length k or less of a given string by using k-truncated DAWGs. For two strings x and y, x is an *absent word* of y iff $x \notin Sub(y)$. An absent word x of y is a *minimal absent word (MAW)* of y if and only if $Sub(x) \setminus \{x\} \subset Sub(y)$. In other words, $x = avb$, where $a, b \in \Sigma$ and $v \in \Sigma^*$, is

Algorithm 3: A pseudo-code of the function $split(u, c)$

```
1  procedure $split(u, c)$
2  $w \leftarrow \delta(u, c)$;
3  make new node $s$;
4  $V \leftarrow V \cup \{s\}$;
5  for each character $a$ such that $\delta(w, a) \neq nil$ do
6      $E \leftarrow E \cup \{(s, a, \delta(w, a))\}$;
7  for each node $t$ such that $\delta(t, a) = w$ and $|\mathrm{long}(t)| \leq |\mathrm{long}(u)|$ do
8      $E \leftarrow E \cup \{(t, a, s)\} \backslash \{(t, a, w)\}$;
9  $L \leftarrow L \cup \{(s, sl(w))\}$;
10 $L \leftarrow L \backslash \{(w, sl(w))\}$;
11 $L \leftarrow L \cup \{(w, s)\}$;
```

Algorithm 4: An $O(\min\{n, k\gamma\} + |MAW_k(y)|)$-time algorithm for computing $MAW_k(y)$

```
   Input: $k$-truncated DAWG $k\text{-}TDAWG(y)$ of $y$.
   Output: All minimal absent words of length up to $k$ for $y$
1  $MAW_k \leftarrow \emptyset$;
2  for each non-source node $u$ of $k\text{-}TDAWG(y)$ do
3      for each character $b$ such that $\delta_{TD}(sl_{TD}(u), b) \neq nil$ do
4          if $\delta_{TD}(u, b) = nil \wedge |\mathrm{long}(sl_{TD}(u))| \leq k - 2$ then
5              $x \leftarrow \mathrm{long}(sl_{TD}(u))$;
6              $MAW_k \leftarrow MAW_k \cup \{axb\}$;          // $(u, a, sl_{TD}(u)) \in L_{k\text{-}TD}$
7  Output $MAW_k$;
```

a MAW of y iff $x \notin Sub(y)$, $av \in Sub(y)$, and $vb \in Sub(y)$. The set of all MAWs of y is denoted by $MAW(y)$. For example, given $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $y = \mathtt{abaac}$, then $MAW(y) = \{\mathtt{aaa}, \mathtt{aab}, \mathtt{bab}, \mathtt{bac}, \mathtt{bb}, \mathtt{bc}, \mathtt{ca}, \mathtt{cb}, \mathtt{cc}\}$.

Given a string y, the following lemma holds for the number of MAWs of y.

Lemma 10 ([63]). *For any string $y \in \Sigma^*$, $\sigma \leq |MAW(y)| \leq (\sigma_y - 1)(|y| - 1) + \sigma$, where $\sigma = |\Sigma|$ and σ_y is the number of distinct characters occurring in y. This bound is tight.*

MAWs can be computed from DAWGs with suffix links in linear time.

Lemma 11 ([32]). *Given a DAWG of string y of length n, $MAW(y)$ can be computed in $O(n + |MAW(y)|)$ time with $O(n)$ working space.*

Let $MAW_k(y)$ denote the MAWs of y which length k or less. For example, given $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and $y = \mathtt{abaac}$, then $MAW_2(y) = \{\mathtt{bb}, \mathtt{bc}, \mathtt{ca}, \mathtt{cb}, \mathtt{cc}\}$. Now we show that $MAW_k(y)$

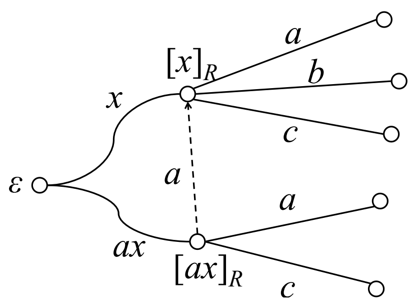


Figure 4.7: Computing minimal absent words from a truncated DAWG.

can be computed from k-$TDAWG(y)$ including its suffix links. Algorithm 4 shows an algorithm for computing $MAW_k(y)$.

Theorem 9. *Given a k-truncated DAWG of string y of length n, Algorithm 4 computes $MAW_k(y)$ in $O(\min\{n, k\gamma\} + |MAW_k(y)|)$ time and $O(\min\{n, k\gamma\})$ working space, where γ is the size of one of the smallest k-attractor of y.*

Proof. First, we show the correctness of our algorithm. For any node u of k-$TDAWG(y)$ where $\mathrm{short}(u)$ is less than k, $EndPos(sl_D(u)) \supset EndPos(u)$ holds because any string in $sl_D(u)$ is a suffix of the strings in u. Thus, if there is an out-edge of u labeled c, there is an out-edge of $sl_D(u)$ labeled c.

Hence, to compute $x = avb$, where $a, b \in \Sigma$ and $v \in \Sigma^*$, such that $x \notin Sub(y)$, $av \in Sub(y)$, and $vb \in Sub(y)$ we need to to find every character b such that there is an out-edge of $v = sl_D(u)$ labeled b but there is no out-edge of u labeled b. The for loop of Line 3 of Algorithm 4 tests all such characters and only those. Hence, Algorithm 4 computes $MAW_k(y)$ correctly.

Figure 4.7 shows k-$TDAWG(y)$. The string ax occurs in y, $ax \not\equiv_E x$ and $[x]_R$ has out-going edges labeled a, b, and c. So xa, xb, and xc occur in y. On the other hand, $[ax]_R$ has out-going edges labeled a and c, but does not b, i.e. axa and axc occur in y and axb does not. Because ax and xb occur in y and axb does not, axb is a minimal absent word of y.

Second, we analyze the time complexity of our algorithm. As mentioned above, the minimal absent words of length 1 for y can be found in $O(n + \sigma)$ time and $O(1)$ working space. By Lemma 10, the σ-term is dominated by the output size $|MAW_k(y)|$. Next, we consider the cost of finding minimal absent words x of length at least 2 and at most k by Algorithm 4.

Let b be any character such that there is an out-edge e of $v = sl_{TD}(u)$ labeled b. There are two cases: (1) If there is no out-edge of u labeled b, then we output a MAW, so we can charge the cost to check it to the output size. (2) If there is an out-edge e' of u labeled b, then we can charge the cost to check e to e'. Since each node u has exactly one suffix link going out from it, each out-edge of u is charged only once in Case (2). Moreover, if the out-edges of every node u and those of $sl_{TD}(u)$ are sorted, we can compute their difference for every node u in $k\text{-}TDAWG(y)$ in overall $O(\min\{n, k\gamma\})$ time. Overall, Algorithm 4 runs in $O(\min\{n, k\gamma\} + |MAW_k(y)|)$ time. The space requirement is clearly $O(\min\{n, k\gamma\})$.

4.5 Truncated DAWGs of Multiple Strings

In this section, we consider truncated DAWG for multiple string. We redefine some notations so they can be used for string sets. Let $S = \{y_1, y_2, \dots, y_m\}$ be a set of strings over Σ. Let $Sub_k(S) = \bigcup_{1=1}^{m} Sub_k(y_i)$. For any string set $S = \{y_1, y_2, \dots, y_m\}$, we define the set of beginning positions $BegPos(x) = \{(i, l) \mid l \in [1, m], i \in [1, |y_l| - |x| + 1], y_l[i..i + |x| - 1] = x\}$ and end positions $EndPos(x) = \{(i, l) \mid l \in [1, m], i \in [|x|, |y_l|], y_l[i - |x| + 1..i] = x\}$ of occurrences of x in S. For any strings s, t, we write $s \equiv_L t$ (resp. $s \equiv_R t$) iff $BegPos(s) = BegPos(t)$ (resp. $EndPos(s) = EndPos(t)$). For any string $x \in \Sigma^*$, the equivalence classes with respect to $\equiv_L$ and $\equiv_R$ that x belongs to, are respectively denoted by $[x]_L$ and $[x]_R$. Also, $\overrightarrow{x}$ and $\overleftarrow{x}$ respectively denote the longest elements of $[x]_L$ and $[x]_R$.

We define the k-truncated DAWG for a set of strings as follows.

Definition 10. *The k-truncated directed acyclic word graph $k\text{-}TDAWG(S)$ for a set S of strings is a directed graph $(V_{k\text{-}TD}, E_{k\text{-}TD})$ such that*

$$\begin{aligned} V_{k\text{-}TD} &= \{[x]_R \mid x \in Sub_k(S)\} \\ E_{k\text{-}TD} &= \{([x]_R, b, [xb]_R) \mid x, xb \in Sub_k(S), b \in \Sigma\}. \end{aligned}$$

Figure 4.8 shows an example of DAWGs and truncated DAWGs of multiple string.

In order to analyze the size of k-truncated DAWG for multiple strings, we introduce a generalized version of k-attractors.

Definition 11. *A k-*attractor Λ *of a string set $S = \{y_1, \dots, y_m\}$ is a set of λ pairs of $l \in [1, m]$ and positions τ of y_l, such that any substring $y_l[i..j]$ with $i \le j < i + k$ of $y_l \in S$ has an occurrence $y_{l'}[i'..j'] = y_l[i..j]$ with $\tau \in [i', j']$ for some $(l', \tau) \in \Lambda$.*

By using the size k-attractors, we show the size of $Sub_k(S)$ as follows.

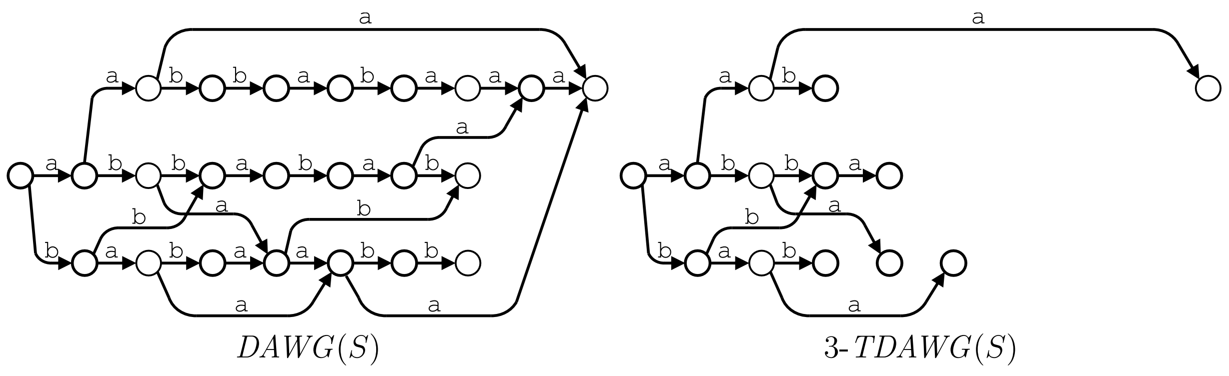


Figure 4.8: The DAWG and 3-truncated DAWG of $S = \{\texttt{abbabab}, \texttt{babaabb}, \texttt{aabbabaaa}\}$.

Lemma 12. *$|Sub_k(S)|$ is in $O(\min\{N, k(\lambda + m)\})$, where N is the total length of all strings in S and λ is the size of one of the smallest k-attractors of S.*

Proof. By the same technique as proof of Lemma 1 in [68], the number of distinct substrings of S with length k is in $O(k\lambda)$. The number of distinct suffixes of $y \in S$ whose length is less than k is at most $(k-1)m$. Therefore, $|Sub_k(S)|$ is in $O(\min\{N, k(\lambda + m)\})$.

From the definition of k-truncated DAWGs, an upper bound of the size of k-$TDAWG(S)$ is shown in the following theorem.

Theorem 10. *The size of k-truncated DAWG k-$TDAWG(S)$ of a set $S = \{y_1, y_2, \ldots, y_m\}$ is $O(\min\{N, k(\lambda + m))$.*

Next, we consider how to construct k-$TDAWG(S)$.

Lemma 13. *Let S and S' be sets of strings such that $S' = S \cup \{y\}$. Let v be a node of both $DAWG(S)$ and $DAWG(S')$. If v does not not exist in k-$TDAWG(S)$, v also does not exist in k-$TDAWG(S')$.*

Proof. Let, $[x]_R^S$ be the equivalence class represented by v in $DAWG(S)$ and $[x]_R^{S'}$ be the equivalence class represented by v in $DAWG(S')$. Assume that v is not in k-$TDAWG(S)$ and in k-$TDAWG(S')$. From the assumption, there is a string w such that $w \notin [x]_R^S$, $w \in [x]_R^{S'}$, and $|w| \leq k$. However, $[x]_R^{S'} \subseteq [x]_R^S$ holds from the definition, which is a contradiction.

By Lemma 13, we can construct k-$TDAWG(S)$ by running Algorithm 1 on all strings in S. Therefore, we get the following theorem on constructing k-$TDAWG(S)$.

Table 4.1: Construction time of DAWGs and truncated DAWGs on artificial and real data in milliseconds

data	DAWG	5-TDAWG	10-TDAWG	20-TDAWG
fib17 ($n = 1597$)	0.158	0.100	0.083	0.068
$(\texttt{ab})^{1000}$	0.199	0.112	0.110	0.092
$(\texttt{abc})^{2000/3}$	0.214	0.113	0.113	0.107
dna.50MB	65.630	10.032	58.196	66.439

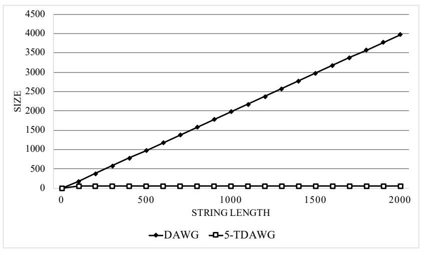
Theorem 11. *Given a set of strings S of size m over an ordered alphabet and a natural number k, we can construct k-TDAWG(S) in $O(N \log \sigma)$ time and $O(\min\{N, k(\lambda + m)\})$ working space in an online manner, where λ is the size of one of the smallest k-attractors of S and N is the total length of all strings in S.*

4.6 Experiments

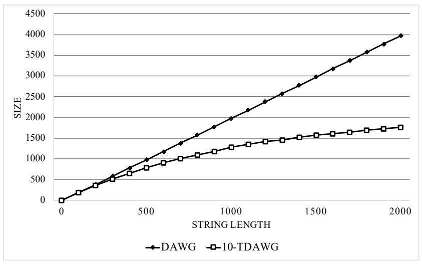
In these experiments, we evaluate the practical performance of truncated DAWGs. First, we compare the number of nodes of DAWG and the number of nodes of truncated DAWG on random strings. Figure 4.9 shows the experimental results on randomly generated strings with the generation rate of each character $P(\texttt{a}) = P(\texttt{b}) = 1/2$. We construct DAWGs and k-truncated DAWGs for $k = 5, 10, 20$. We can confirm that the size of k-truncated DAWGs is less than or equal to the size of DAWGs. When $k = 5, 10$, we can see that the size of k-truncated DAWG is much smaller than DAWGs.

Figure 4.10 shows the experimental results on randomly generated strings with the generation rate of each character $P(\texttt{a}) = 3/4$ and $P(\texttt{b}) = 1/4$. The generated strings are expected to be more repetitive. We construct DAWGs and k-truncated DAWGs for $k = 5, 10, 20$. If we compare Figure 4.9 and Figure 4.10, we can see that the size of k-truncated DAWGs for repetitive string are smaller than the size of k-truncated DAWGs for non-repetitive string.

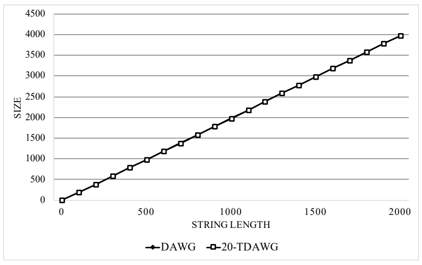
Last, we compare the construction time of DAWGs and truncated DAWGs on artificial and real data. We use Fibonacci strings (fib17), $(\texttt{ab})^{1000}$, and $(\texttt{abc})^{2000/3}$ for artificial data, and dna.50MB from Pizza&Chili Corpus [1] for real data. As we can see from Table 4.1, truncated DAWGs are constructed faster than DAWGs, especially 5-TDAWG on dna.50MB data. We can conclude that using truncated DAWGs is effective when we want to process short substrings on very big string data.



$k = 5$

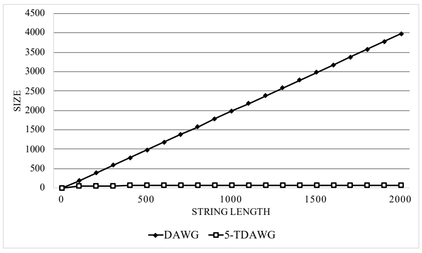


$k = 10$

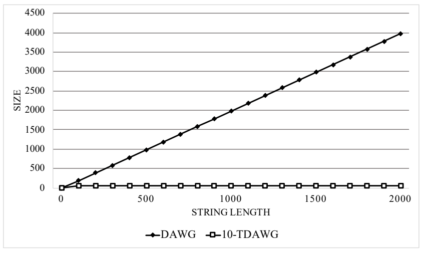


$k = 20$

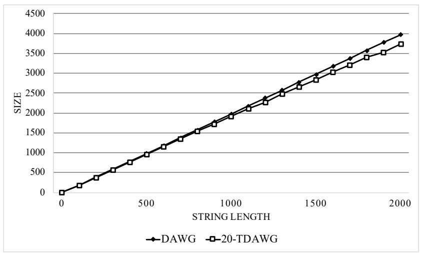
Figure 4.9: Comparison of the number of nodes of DAWGs and k-truncated DAWGs on random strings $P(\mathtt{a}) = 1/2, P(\mathtt{b}) = 1/2$.



$k = 5$



$k = 10$



$k = 20$

Figure 4.10: Comparison of the number of nodes of DAWGs and k-truncated DAWGs on repetitive random strings $P(\mathtt{a}) = 3/4, P(\mathtt{b}) = 1/4$.

4.7 Conclusion and Discussion

In this chapter, we proposed a new data structure called truncated DAWG. We show that the k-truncated DAWG of y, denoted by $k\text{-}TDAWG(y)$, is a subgraph of $DAWG(y)$, and can be stored in $O(\min\{n, k\gamma\})$ space. We also presented an $O(n \log \sigma)$ time and $O(\min\{n, k\gamma\})$ space algorithm for constructing $k\text{-}TDAWG(y)$, where n is the length of y, σ is the alphabet size, and γ is the size of one of the smallest k-attractors of y. As an application of $k\text{-}TDAWG(y)$, we presented an $O(\min\{n, k\gamma\} + |MAW_k(y)|)$ time algorithm to compute the set $MAW_k(y)$ of all minimal absent words of y whose size is smaller than or equal to k by using $k\text{-}TDAWG(y)$.

In addition, we also presented k-truncated DAWGs for multiple strings. For a set of strings S, We showed that the size of $k\text{-}TDAWG(S)$ is $O(\min\{N, k(\lambda + m)\})$, where N is the total length of all patterns in S, m is the number of patterns in S, and λ is the size of one of the smallest k-attractors of S. We also showed that $k\text{-}TDAWG(S)$ can be constructed and used in the same manner as that for the k-truncated DAWG of a single string.

Last, the experimental results show that the size of k-truncated DAWGs are much smaller than the size of DAWGs when k is small and when the input string is repetitive. Moreover, the experimental results show that k-truncated DAWGs can be constructed faster than DAWGs practically. We can conclude that MAW_k can be computed faster and in smaller memory by using k-truncated DAWGs.

Our future work is to find a way to construct truncated DAWGs in a faster time and smaller space by using compressed strings such as LZ77 factorization. We are also interested in truncated version of compressed directed acyclic word graphs [12] (CDAWGs). Since CDAWGs use less space than suffix trees and DAWGs, truncated CDAWGs are expected to use less memory than truncated suffix trees and truncated DAWGs.

Chapter 5

An Improved Data Structure for Left-right Maximal Generic Words Problem

String Data Mining is an important research area which has received special attention. One of the fundamental tasks in this area is the frequent pattern mining, the aim of which is to find patterns occurring in at least d documents in D for a given collection D of documents and a given threshold d, where the patterns are drawn from a fixed hypothesis space. The task is useful not only in extracting patterns which characterize the documents in D, but also in enumerating candidates for the most classificatory pattern that separates two given sets of strings. The hypothesis space varies depending upon users' particular interest or purpose, and is ranging from the substring patterns to the VLDC patterns. Frequent substring patterns are often referred to as *generic words*. The generic words mining problem (or the frequent substring pattern mining problem) has a wide variety of applications, e.g., Computational Biology, Text mining, and Text Classification [51, 11, 36].

One interesting variant of the generic words mining problem is the *right maximal generic words problem*, formulated by Kucherov et al. [51]. In this variant, a pattern p is given as additional input, which limits the outputs to the right extensions of p. Moreover, the outputs are limited to the *maximal* ones. Formally, the problem is to preprocess D so that, for any pattern p and for any threshold d, all right extensions of p that are d-right maximal can be computed efficiently, where a string w is said to be *d-right maximal* if x occurs in at least d documents but xa occurs in less than d documents for any character a. They presented in [51] an $O(n)$-size data structure which answers queries in $O(|p| + r)$ time, where n is the total length of strings in D and r is the number of outputs. Later, Biswas et al. [11] developed a succinct data structure of size $n \log |\Sigma| + o(n \log |\Sigma|) + O(n)$ bits of space, which answers queries in $O(|p| + \log \log n + r)$

time.

As a generalization, Nishimoto et al. [67] defined the *left-right-maximal generic word problem*. In this problem, all superstrings of p that are d-left-right maximal should be answered, where a string w is said to be *d-left-right maximal* if x has a document frequency $\geq d$ but xa and ax respectively have a document frequency $< d$ for any character a.

One naive solution to this problem is to compute the sets M_d of d-left-right maximal strings for $1 \leq d \leq m$, where m is the number of documents in D and then apply the optimal algorithm of Muthukrishnan [64] for the document listing problem, regarding M_d as input document collection. The query time is $O(|p| + o)$ time, where o is the number of outputs. The space requirement is $O(n^2 \log m)$ since the Muthukrishnan algorithm uses the (generalized) suffix tree of input document collection and the size of suffix tree for M_d can be shown to be $O(n^2/d)$ for every $d = 1, \ldots, m$. The $O(n^2 \log m)$ space requirement is, however, impractical when dealing with a large-scale document collection.

In [67] Nishomoto et al. presented an $O(n \log n)$-space data structure which answers queries in $O(|p| + r \log n + o \log^2 n)$ time, where r is the number of d-right-maximal strings that subsume p as a prefix. The factor $O(r \log n)$ is for computing the d-right-maximal right extensions of p, which are required for computing d-left-right-maximal extensions of p in their method.

In this chapter, we address the left-right-maximal generic word problem and propose an $O(n \log m)$-space data structure with query time $O(|p| + o \log \log m)$. The data structure outperforms the previous work by Nishimoto et al. [67] both in the query time and in the space requirement.

Our method uses the suffix trees of M_d for $d = 1, \ldots, m$. For a string set $S = \{w_1, \ldots, w_\ell\}$, Usually, "the suffix tree of S" means the suffix tree of $\{w_1\$_1, \ldots, w_\ell\$_\ell\}$ with ℓ distinct endmarkers $\$_1, \ldots, \$_\ell$, or the suffix tree of $S\$ = \{w_1\$, \ldots, w_\ell\$\}$ with a single endmarker \$. In both cases, the size of suffix tree is proportional to the total length of the strings in S. The total size of suffix trees of $M_d\$$ for $d = 1, \ldots, m$ is $O(nm)$, where n is the total length of D. Our idea in reducing the space requirement is to replace the suffix tree of $M_d\$$ with the suffix tree of M_d. Removing the endmarker successfully reduces the $O(nm)$ total size of the suffix trees to $O(n \log m)$, with a small sacrifice of query time.

5.1 Notations for Generic Words and Tools

Let $Pre(S) = \bigcup_{w \in S} Pre(w)$, $Sub(S) = \bigcup_{w \in S} Sub(w)$ and $Suf(S) = \bigcup_{w \in S} Suf(w)$ for any set S of strings. The *reversal* of a string w, denoted by w^R, is defined to be $w[|w|] \dots w[1]$. Let $S^R = \{w^R \mid w \in S\}$ for any set S of strings.

The *longest repeating suffix* of a string x is the longest suffix of x that occurs elsewhere in x. Let $LRS(x)$ denote the length of the longest repeating suffix of x. We note that any suffix of x longer than $LRS(x)$ occurs only once in x.

5.1.1 d-left-right maximality of strings

Let D be a set of documents (strings). The *document frequency* of a string x in D, denoted by $df_D(x)$, is defined to be the number of documents in D that contain x as a substring. We write $df(x)$ instead of $df_D(x)$ when D is clear from the context.

A string x is said to be *d-left maximal* w.r.t. D if $df(x) \geq d$ and $df(ax) < d$ for all $a \in \Sigma$, and said to be *d-right maximal* w.r.t. D if $df(x) \geq d$ and $df(xa) < d$ for all $a \in \Sigma$. A string x is said to be *d-left-right maximal* w.r.t. D if it is d-left maximal and d-right maximal w.r.t. D. Let M_d denote the sets of d-left-right maximal strings w.r.t. D.

Example 1. *For $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aabababbaa}, \texttt{abaababbba}\}$, the sets of d-left-right maximal strings for $d = 1, 2, 3, 4$ are as follows: $M_1 = D$, $M_2 = \{\texttt{aaabaab}, \texttt{aabab}, \texttt{abaaba}, \texttt{ababb}, \texttt{abbba}\}$, $M_3 = \{\texttt{aaba}, \texttt{abaab}, \texttt{abb}, \texttt{bba}\}$ and $M_4 = \{\texttt{aaba}, \texttt{baa}\}$.*

Lemma 14 ([51])**.** *For any set D of strings with total length n, the number of d-right maximal strings w.r.t. D is $O(n/d)$.*

Lemma 15. *For any string y the following statements hold.*

1. *Let z be the shortest string such that $yz \in Suf(M_d)$. If $xyz \in M_d$ for some string x, then xy is d-left maximal.*

2. *Let x be the shortest string such that $xy \in Pre(M_d)$. If $xyz \in M_d$ for some string z, then yz is d-right maximal.*

Proof. It suffices to give proof only for the first statement. Suppose to the contrary that xy is not d-left maximal. Then, $df(xy) \geq df(xyz) \geq d$, and there exists some $\alpha \in \Sigma^+$ such that αxy is d-left maximal. Since xyz is d-maximal, $df(\alpha xyz) < d$. Furthermore, since

$df(\alpha xy) \geq d$, there exists a prefix z' of z such that $\alpha xyz'$ is d-maximal and $|z'| < |z|$. This implies $yz' \in Suf(M_d)$ and contradicts that z is the shortest such string. Therefore, xy must be d-left maximal.

5.1.2 Suffix trees for a string set

Let $S = \{w_1, \ldots, w_\ell\}$ be a set of nonempty strings with total length n. The suffix tree [75] of S, denoted by $STree(S)$, is a path-compressed trie which represents all suffixes of S. More formally, $STree(S)$ is an edge-labeled rooted tree such that (1) Every internal node is branching; (2) The out-going edges of every internal node begin with mutually distinct characters; (3) Each edge is labeled by a non-empty substring of S; (4) For each suffix s of S, there is a unique path from the root which spells out s and the path possibly ends on an edge.

It follows from the definition of $STree(S)$ that the numbers of nodes and edges in $STree(S)$ are $O(n)$, respectively. By representing every edge label x by a triple (i, j, k) of integers such that $x = w_k[i..j]$, $STree(S)$ can be represented in $O(n)$ space. The *size* of suffix tree $STree(S)$ is defined to be the number of nodes and is denoted by $|STree(S)|$.

A node v of $STree(S)$ is said to *represent* a string x if the path from the root to v spells out x. For a substring x of S, the *locus* of x in $STree(S)$ is defined to be the highest node v that represents a right extension of x. A string x is said to be *explicit* in $STree(S)$ if there exists a node v of $STree(S)$ that represents x and *implicit* otherwise.

In this chapter, we properly use the suffix trees of the following three types to suit its use.

1. $STree(\overline{S})$ where $\overline{S} = \{w_1\$_1, \ldots, w_\ell\$_\ell\}$ and $\$_1, \ldots, \$_\ell$ are mutually distinct endmarkers not in Σ.

2. $STree(S\$)$ where $\$$ is an endmarker not in Σ.

3. $STree(S)$ without endmarker.

The above suffix trees are all capable of determining whether $x \in Sub(S)$ for any $x \in \Sigma^+$. $STree(S)$ cannot distinguish the elements of $Suf(S)$ from those of $Sub(S)$ whereas $STree(S\$)$ and $STree(\overline{S})$ can determine whether $x \in Suf(S)$ for any $x \in Sub(S)$. In addition, $STree(\overline{S})$ can determine the set of indices k such that $x \in Suf(w_k)$. It is easy to see that:

Lemma 16. $|STree(\overline{S})| \geq |STree(S\$)| \geq |STree(S)|$ *for any set S of strings.*

5.1.3 Tools

Let x be a fixed string over $A = \{1, \ldots, \sigma\}$. The Rank query $rank_x(a, i)$ returns the number of occurrences of $a \in A$ in the prefix $x[..i]$ of x, and the Select query $select_x(a, j)$ returns the position of j-th occurrence of $a \in A$ in x.

Lemma 17 ([38]). *There is an $O(|x|)$ space data structure that answers Rank/Select queries in $O(\log \log \sigma)$ time.*

Let T be an ordered tree with n nodes and with function val that maps the nodes to the integers. The find-less-than (FLT) query on tree T is, given a threshold τ and a node v of T, to enumerate the descendants u of v with $val(u) < \tau$.

Lemma 18. *We can build from T an $O(n)$ space data structure in $O(n)$ time that answers FLT queries in $O(out)$ time, where out is the number of outputs.*

Proof. Let $v_1, \ldots, v_n$ be the nodes T in the preorder. Let B be an array such that $B[i] = val(v_i)$ for all $i \in [1..n]$. Then, the problem of FLT queries on tree can be reduced to the problem of FLT queries on array B defined as follows:

> Given a threshold τ and a subinterval $[i..j]$ of $[1..n]$, enumerate the indices k in $[i..j]$ such that $val(B[k]) < \tau$.

FLT queries on array B of size n can be answered in linear time proportional to the number of outputs, by repeated use of the Range Minimum Query (see [64]).

5.2 Main Result and Algorithm Outline

5.2.1 Main result

Our problem is formulated as follows.

Problem 1.

To-preprocess: *A subset $D = \{w_1, \ldots, w_m\}$ of Σ^+.*

Query: *A string $p \in \Sigma^+$ and an integer $d \in [1..m]$.*

Answer: *The strings in $\Sigma^* p \Sigma^* \cap M_d$.*

One naive solution to the problem would be to apply the optimal algorithm of Muthukrishnan et al. [64] for the document listing problem regarding M_d as input document collection. This solution requires space proportional to the total size of suffix trees $STree(\overline{M_d})$ for $d = 1, \dots m$. The following lemmas hold for the size of the suffix trees.

Lemma 19. *The suffix trees $STree(M_d\$)$ and $STree(M_d)$ are of size $O(n)$ for any $d = 1, \dots, m$, and the suffix tree $STree(\overline{M_d})$ is of size $O(n^2/d)$ for any $d = 1, \dots, m$.*

Proof. First, we show that $STree(M_d\$)$ has $O(n)$ leaves. Let v be any leaf of $STree(M_d\$)$, and let $x\$$ be the string represented by v. There is a string α such that $\alpha x \in M_d$. Assume, for a contradiction, that x is implicit in $STree(\overline{D})$. Then, there uniquely exists a character a such that every occurrence of x in the strings of $\overline{D}$ is followed by a. This contradicts $\alpha x \in M_d$. Hence x is explicit in $STree(\overline{D})$. The number of leaves of $STree(M_d\$)$ is not greater than the number of nodes of $STree(\overline{D})$, which is $O(n)$. By Lemma 16, $STree(M_d\$)$ and $STree(M_d)$ are of size $O(n)$. Next, we prove that $STree(\overline{M_d})$ has $O(n^2/d)$ leaves. Let v be any leaf of $STree(\overline{M_d})$, and let $x\$_i$ be the string represented by v. As the previous discussion, x is explicit in $STree(\overline{D})$. There are $|M_d|$ endmarkers $\$_j$ in $\overline{M_d}$, and by Lemma 14 we have $|M_d| = O(n/d)$. Hence the number of leaves of $STree(\overline{M_d})$ is not greater than $O(n/d)$ times the number of nodes of $STree(\overline{D})$, which is $O(n^2/d)$.

Lemma 20. *For any natural number d and m with $1 \leq k \leq m$, there exists string set D such that the total length of strings in M_d is $\omega(n/d)$.*

Proof. (σ, k)-*de Bruijn sequence* is a minimum length string on the alphabet set whose size is σ which contains all k-mers exactly once and of length is $\sigma^k + k - 1$ [26]. A set of (σ, k)-de Bruijn sequences is called *orthogonal* if no two of them contain the same $(k+1)$-mer. Lin et al. showed that for any $k > 0$, there exists an orthogonal set of (σ, k)-de Bruijn sequences with size at least $\lfloor \sigma/2 \rfloor$[53]. For given m, d, consider the following string set D. Let D be the union of the orthogonal set of $(2d, k)$-de Bruijn sequences of size d with $d \geq 2$ and $m - d$ strings with length 1. Then $n = m((2d)^k + k - 1 + m - d)$ and $n/d = (2d)^k + k - 1 + m - d$. Since M_d is the set of all k-mers, the total length of strings in M_d is $k(2d)^k$. Because $k \in \Theta(\log_d(n/d))$ and $d \ll n$, the total length of strings in M_d is in $\omega(n/d)$.

Example 2 is the example of Lemma 20.

Example 2. *When $m = 5, d = 2, k = 2$, then $D = \{\texttt{a}, \texttt{a}, \texttt{a}, \texttt{aabbccddbdadcacba}, \texttt{aaddccbbabdbcdaca}\}$, $M_2 = \{\texttt{aa}, \texttt{ab}, \texttt{ac}, \texttt{ad}, \texttt{ba}, \texttt{bb}, \texttt{bc}, \texttt{bd}, \texttt{ca}, \texttt{cb}, \texttt{cc}, \texttt{cd}, \texttt{da}, \texttt{db}, \texttt{dc}, \texttt{dd}\}$.*

The naive solution answers queries in $O(|p| + o)$ time using $O(n^2 \log m)$ space, where o is the number of outputs. The $O(n^2 \log m)$ space requirement is, however, impractical for dealing with a large-scale document set.

Our solution reduces the $O(n^2 \log m)$ space requirement to $O(n \log m)$ with a little sacrifice in query response time.

Theorem 12. *There exists an $O(n \log m)$ space data structure for Problem 1 which answers queries in $O(|p| + o \log \log m)$ time, where o is the number of outputs.*

5.2.2 Algorithm outline

Our task is, given a string $p \in \Sigma^+$, to enumerate the strings $\alpha p \beta$ in M_d with $\alpha, \beta \in \Sigma^*$. One solution would be to enumerate the strings αp in $Pre(M_d)$ with $\alpha \in \Sigma^*$, and then, for each αp enumerate the strings $\alpha p \beta$ in M_d with $\beta \in \Sigma^*$. The resulting enumeration, however, contains duplicates if there is some string in M_d containing p more than once. Consider the string abaaba which contains $p = \texttt{ab}$ twice in Example 3. The strings ab ($\alpha = \varepsilon$) and abaab ($\alpha = \texttt{aba}$) appear in the enumeration of αp, and therefore the string abaaba appears twice in the enumeration of $\alpha p \beta$.

Example 3. *Let $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aabababbaa}, \texttt{abaababbba}\}, d = 2$ and $p = \texttt{ab}$. Then M_2 is $\{\texttt{aaabaab}, \texttt{aabab}, \texttt{abaaba}, \texttt{ababb}, \texttt{abbba}\}$ and the answer is $\{\texttt{aaabaab}, \texttt{aabab}, \texttt{abaaba}, \texttt{ababb}, \texttt{abbba}\}$. (1) $\Sigma^* p \cap Pre(M_d) = \{\texttt{aaab}, \texttt{aaabaab}, \texttt{aab}, \texttt{aabab}, \texttt{ab}, \texttt{abaab}, \texttt{abab}\}$. (2) Their d-left-right-maximal extensions are $\{\texttt{aaabaab}\}$, $\{\texttt{aaabaab}\}$, $\{\texttt{aabab}\}$, $\{\texttt{aabab}\}$, $\{\texttt{abaaba}, \texttt{ababb}, \texttt{abbba}\}$, $\{\texttt{abaaba}\}$, $\{\texttt{ababb}\}$, respectively. (3) The union of these string sets is $\{\texttt{aaabaab}, \texttt{aabab}, \texttt{abaaba}, \texttt{ababb}, \texttt{abbba}\}$, which coincides with the answer.*

In order to avoid such duplicates in enumeration, we put a restriction on the enumeration of the strings $\alpha p \in Pre(M_d)$. That is, we enumerate the strings $\alpha p \in Pre(M_d)$ satisfying the condition that αp contains p just once, which can be replaced with $LRS(\alpha p) < |p|$. The outline of our algorithm is as follows:

Step 1. Enumerate the strings αp such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$ and $LRS(\alpha p) < |p|$.

Step 2. For each string αp obtained in Step 1, enumerate the strings $\alpha p\beta$ such that $\beta \in \Sigma^*$ and $\alpha p\beta \in M_d$.

Example 4. *Let* $D = \{\mathtt{aaabaabaaa}, \mathtt{aaabaabbba}, \mathtt{aababab baa}, \mathtt{abaababbba}\}, d = 2$ *and* $p = \mathtt{ab}$*. Then* M_2 *is* $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$ *and the answer is* $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$*. (1)* $\Sigma^* p \cap Pre(M_d) = \{\mathtt{aaab}, \mathtt{aaabaab}, \mathtt{aab}, \mathtt{aabab}, \mathtt{ab}, \mathtt{abaab}, \mathtt{abab}\}$*. (2) Of the seven strings, the three strings* aaab*,* aab*,* ab *satisfy the condition* $LRS(x) < |p|$*. Their d-right extensions are* $\{\mathtt{aaabaab}\}$*,* $\{\mathtt{aabab}\}$*,* $\{\mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$*, respectively. These sets are mutually disjoint. (3) The union of the disjoint sets is* $\{\mathtt{aaabaab}, \mathtt{aabab}, \mathtt{abaaba}, \mathtt{ababb}, \mathtt{abbba}\}$*, which coincides with the answer (see Figure 5.1).*

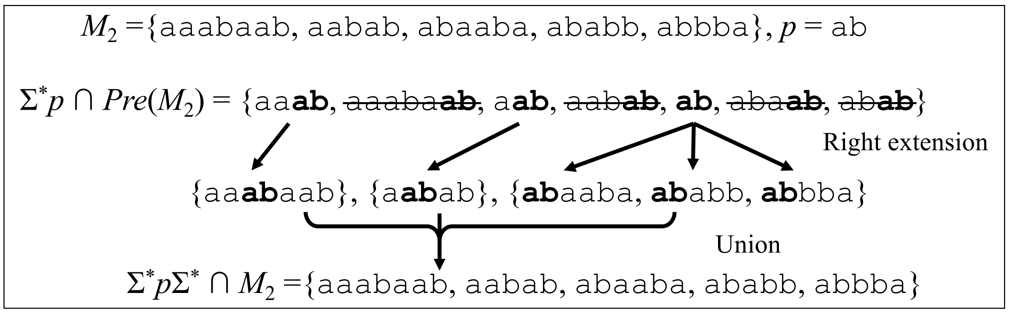

M_2 ={aaabaab, aabab, abaaba, ababb, abbba}, p = ab
$\Sigma^* p \cap Pre(M_2)$ = {aa**ab**, ~~aaabaa**ab**~~, a**ab**, ~~aab**ab**~~, **ab**, ~~abaa**ab**~~, ~~ab**ab**~~}
Right extension
{aa**ab**aab}, {a**ab**ab}, {**ab**aaba, **ab**abb, **ab**bba}
Union
$\Sigma^* p \Sigma^* \cap M_2$ ={aaabaab, aabab, abaaba, ababb, abbba}

Figure 5.1: Illustration of Example 4.

5.3 Simplified Solution

For the sake of simplicity in presentation, we here present a simplified version of our algorithm using an $O(nm)$ space data structure which answers queries in $O(|p| + o)$ time, where o is the number of outputs. How to improve the data structure will be described in the next section.

Basically, we represent substrings of M_d as their loci in $STree(M_d)$. We note that although the strings αp in Step 1 may be represented as implicit nodes of $STree(M_d)$, using their loci does not affect the result of Step 2. The algorithm outline can then be rewritten as follows.

Step 1. Enumerate the loci v of αp in $STree(M_d)$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$ and $LRS(\alpha p) < |p|$.

Step 2. For each locus v obtained in Step 1, enumerate the loci of $x\beta$ in $STree(M_d)$ such that $\beta \in \Sigma^*$ and $x\beta \in M_d$, where x is the string represented by v in $STree(M_d)$.

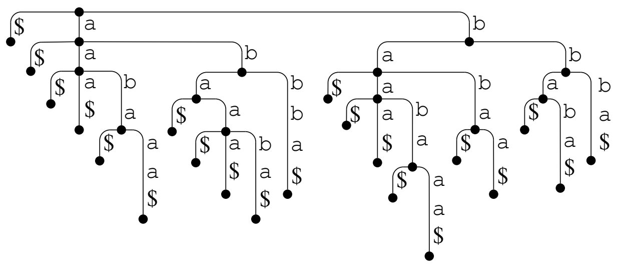


Figure 5.2: Suffix tree $STree({M_2}^R\$)$.

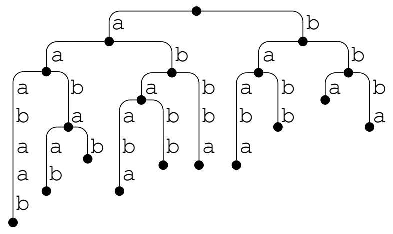


Figure 5.3: Suffix tree $STree(M_2\$)$.

5.3.1 Implementation of Step 1

We use the suffix trees $STree({M_d}^R\$)$ for $d = 1, \dots, m$. We note that there is a natural one-to-one correspondence between the strings x in $Pre(M_d)$ and the leaves of $STree({M_d}^R\$)$ representing $x^R\$$. We also note that for any $p \in \Sigma^+$, the strings in $Pre(M_d) \cap \Sigma^* p$ correspond to the leaves of the subtree rooted at the locus v of p^R in $STree({M_d}^R\$)$. Of the leaves representing $x^R\$$, we have to select those satisfying $LRS(x) < |p|$.

In the running example, the leaves of the subtree rooted at the locus of $p^R = \texttt{ba}$ in $STree({M_2}^R\$)$ represent the strings ba\$, baa\$, baaa\$, baaba\$, baabaaaa\$, baba\$, babaa\$. Of the seven strings of the form $x^R\$$, the three strings ba\$, baa\$, baaa\$ satisfy the condition $LRS(x) < |p|$.

Define the function val from the nodes of $STree({M_d}^R\$)$ to the integers by: For any node u of $STree({M_d}^R\$)$, let $val(u) = LRS(x)$ if u is a leaf of $STree({M_d}^R\$)$, and ∞ otherwise, where x is the string such that u represents x^R. By applying the FLT query technique, mentioned in Section 5.1.3, to the tree $STree({M_d}^R\$)$ with val, we can compute the leaves of $STree({M_d}^R\$)$ representing $(\alpha p)^R\$$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$ and $LRS(\alpha p) < |p|$. From such a leaf, we can obtain the locus of αp in $STree(M_d)$ in constant time by keeping pointers from the nodes u of $STree({M_d}^R\$)$ to the loci of x in $STree(M_d)$, where x is the string such that x^R is represented by u in $STree({M_d}^R\$)$.

5.3.2 Implementation of Step 2

We note that the locus of any string in M_d is a leaf of $STree(M_d)$. The outputs of Step 2 are thus the leaves u of the subtree rooted at v representing strings in M_d. Define the function val from the nodes of $STree(M_d)$ to the integers by: For any node u of $STree(M_d)$, let $val(u) = 0$ if u is a leaf and represents some string in M_d, and 1 otherwise. We again apply the FLT query technique to the tree $STree(M_d)$ with val, to enumerate the loci of $x\beta$ appropriately.

5.3.3 Query time and space requirement

In Step 1, computing the locus of p^R in $STree({M_d}^R\$)$ takes $O(|p|)$ time. Each execution of the FLT query takes constant time in Steps 1 and 2. Thus the query time is $O(|p| + o)$, where o is the number of outputs. For $d = 1, \ldots, m$, the suffix trees $STree(M_d)$ and $STree({M_d}^R\$)$, and the relevant data structures for the FLT queries require $O(n)$ space. The total space of our data structure is $O(nm)$.

5.4 Space Efficient Implementation of Step 1

As seen in Section 5.3.3, the use of the suffix trees $STree({M_d}^R\$)$ for $d = 1, \ldots, m$ in Step 1 causes the $O(nm)$ space requirement. Our idea to reduce the space requirement is to substitute $STree({M_d}^R)$ for $STree({M_d}^R\$)$. The following lemma gives an upper bound on the total size of suffix trees $STree({M_d}^R)$.

Lemma 21. *The suffix trees $STree(M_d)$ for $d = 1, \ldots, m$ are, respectively, of size $O(n/d)$, and their total size is $O(n \log m)$.*

Proof. It suffices to show that $STree(M_d)$ has $O(n/d)$ leaves. Let v be any leaf of $STree(M_d)$, and let x be the string represented by v. Assume, for a contradiction, that x is not d-right maximal. Then, there exists a string $\beta \in \Sigma^+$ such that $\alpha x\beta \in M_d$ for some $\alpha \in \Sigma^*$. Thus $x\beta$ is a suffix of M_d, which contradicts that v is a leaf of $STree(M_d)$. Therefore x is d-right maximal. The number of leaves of $STree(M_d)$ is not greater than the number of d-right maximal strings, which is $O(n/d)$ by Lemma 14.

The difficulty in using not $STree({M_d}^R\$)$ but $STree({M_d}^R)$ is that the string $(\alpha p)^R$ is possibly implicit in $STree({M_d}^R)$ whereas the string $(\alpha p)^R\$$ is necessarily explicit and represented by a leaf in $STree({M_d}^R\$)$. We partition Step 1 into two parts:

Step 1A. Enumerate the loci of αp in $STree(M_d)$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$, $LRS(\alpha p) < |p|$ and $(\alpha p)^R$ is explicit in $STree({M_d}^R)$.

Step 1B. Enumerate the loci of αp in $STree(M_d)$ such that $\alpha \in \Sigma^*$, $\alpha p \in Pre(M_d)$, $LRS(\alpha p) < |p|$ and $(\alpha p)^R$ is implicit in $STree({M_d}^R)$.

Step 1A can be done in $O(|p| + o)$ time with $O(n \log m)$ space in almost the same way as Section 5.3.1. Below we describe how to implement Step 1B.

5.4.1 Implementation of Step 1B

Lemma 22. *For any string x in $Pre(M_d)$, x^R is explicit in $STree(D^R\$)$.*

Proof. Let $\beta \in \Sigma^*$ be a string such that $x\beta \in M_d$. Since the string $(x\beta)^R$ is d-left-right maximal, it is explicit in $STree(D^R\$)$ and therefore its suffix x^R is also explicit in $STree(D^R\$)$.

We thus use $STree(D^R\$)$ to represent strings in $Pre(M_d)$.

Let q_1 and q_2 be the strings represented by the loci of p^R in $STree(D^R\$)$ and in $STree({M_d}^R)$, respectively. The *p-critical* path of $STree(D^R\$)$ is the path from u_1 to u_2 such that u_1 and u_2 are the nodes of $STree(D^R\$)$ representing q_1 and q_2, respectively. A string x and the node representing x^R in $STree(D^R\$)$ are said to be *p-satisfying* if x is a left extension of p such that $x \in Pre(M_d)$, $LRS(x) < |p|$ and x^R is implicit in $STree({M_d}^R)$. An edge e of $STree({M_d}^R)$ and the path corresponding to e in $STree(D^R\$)$ are said to be *p-admissible* if e is in the subtree rooted at the node representing q_2 and at least one implicit node is present on e which represents the reversal x^R of a p-satisfying string x.

Every p-satisfying node of $STree(D^R\$)$ is present on: (i) the p-critical path of $STree(D^R\$)$ or (ii) a p-admissible path of $STree({M_d}^R)$. Thus, the enumeration of the loci of αp in $STree(M_d)$ can be performed as follows.

(1) Enumerate the p-admissible paths of $STree(D^R\$)$.

(2) For each p-admissible path of $STree(D^R)$ and for the p-critical path of $STree(D^R)$, enumerate the p-satisfying nodes on it.

(3) For each p-satisfying node of $STree(D^R\$)$ representing x^R, compute the locus of x in $STree(M_d)$.

Example 5. *Suppose that* $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aababababbaa}, \texttt{abaababbba}\}$, $d = 2$ *and* $p = \texttt{abab}$*. Then,* $q_1 = \texttt{baba}$ *and* $q_2 = \texttt{babaaa}$ *(see Figure 5.4). We want to compute* $\texttt{baba}$ *in Step 1B.*

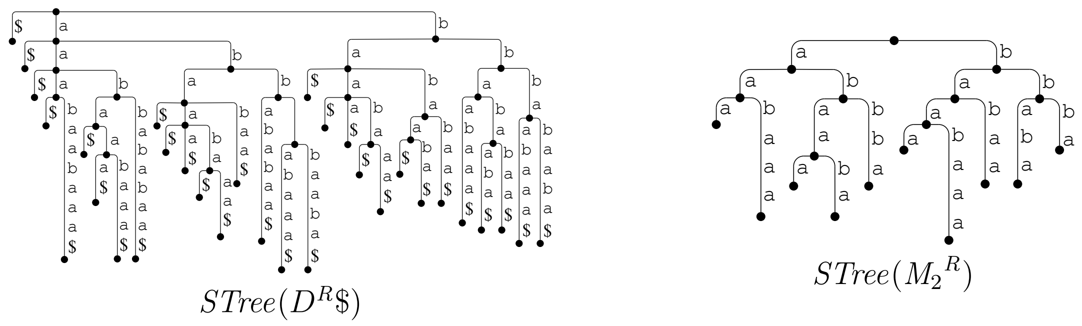


Figure 5.4: $STree(D^R\$)$ and $STree({M_2}^R)$ for $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aababababbaa}, \texttt{abaababbba}\}$.

In (1), we shall enumerate all p-admissible edges of $STree({M_d}^R)$. With each edge (s, t) of $STree({M_d}^R)$, we associate the value $LRS(z^R)$ such that x and y are the strings represented by s and t, respectively, and $z = y[..i]$ where i is the smallest integer in $[|x| + 1, |y| - 1]$ with $z \in Suf({M_d}^R)$ (i.e. $z^R \in Pre(M_d)$). We associate ∞ with it, if no such i exists. Then, we can enumerate all p-admissible edges of $STree({M_d}^R)$, by applying the FLT query technique to $STree({M_d}^R)$, with regarding the value associated with the incoming edge (s, t) of a node t as the value of t. Computing the loci of p^R in $STree({M_d}^R\$)$ and $STree(D^R\$)$ takes $O(|p|)$ time. Execution of the FLT query takes constant time.

In (2), we proceed to examine nodes representing x^R on the path until we encounter a node representing x^R with $LRS(x) \geq |p|$, by repeatedly querying with the data structure stated in the following lemma.

Lemma 23. *There exists an* $O(n \log m)$ *size data structure which, given a node of* $STree(D^R\$)$ *representing string* y^R*, returns in* $O(\log \log m)$ *time the locus of* $(xy)^R$ *in* $STree(D^R\$)$ *such that* x *is the shortest string with* $xy \in Pre(M_d)$*, and* nil *if no such* x *exists.*

In (3), for each p-satisfying node of $STree(D^R\$)$ representing x^R, compute the locus of x in $STree(M_d)$ by using the data structure stated in the following lemma.

Lemma 24. *The locus of a string* x *in* $STree(M_d)$ *can be computed in* $O(\log \log m)$ *time from the locus of* x^R *in* $STree(D^R\$)$ *using an* $O(n \log m)$ *space data structure.*

The suffix tree $STree(D^R\$)$ takes $O(n)$ space. For $d = 1, \ldots, m$, the suffix trees $STree(M_d)$ and $STree({M_d}^R\$)$, and the relevant data structures for the FLT queries require $O(n)$ space. The total computation time of Step 1B is $O(|p| + o \log\log m)$ and the total space of our data structure is $O(n \log m)$.

5.4.2 Proofs of Lemmas 23 and 24

To complete the proof of Theorem 12, we give proofs of Lemmas 23 and 24. For the sake of convenience, we first prove Lemma 24.

Proof of Lemma 24

From the locus of x^R in $STree(D^R\$)$ we can obtain the locus of x in $STree(D\$)$ in constant time by using direct links from the nodes of $STree(D^R\$)$ to the corresponding nodes of $STree(D\$)$. Thus we describe how to compute from the locus of x in $STree(D\$)$ the locus of x in $STree(M_d)$ in $O(\log\log m)$ time using $O(n \log m)$ space.

A node v of $STree(D\$)$ representing string z is called a d-node if z is explicit in $STree(M_d)$. The locus of x in $STree(M_d)$ then corresponds to the earliest d-node preceded by the locus of x in the pre-order traversal of $STree(D\$)$.

Example 6. *Suppose that* $D = \{\texttt{aaabaabaaa}, \texttt{aaabaabbba}, \texttt{aabababbaa}, \texttt{abaababbba}\}$, $d = 2$ *and* $x = \texttt{aab}$. *Then the earliest* 2*-node preceded by the locus of* x *is the node representing* aaba *(see Figure 5.5). The locus of* x *in* $STree(M_2)$ *represents the same string* aaba.

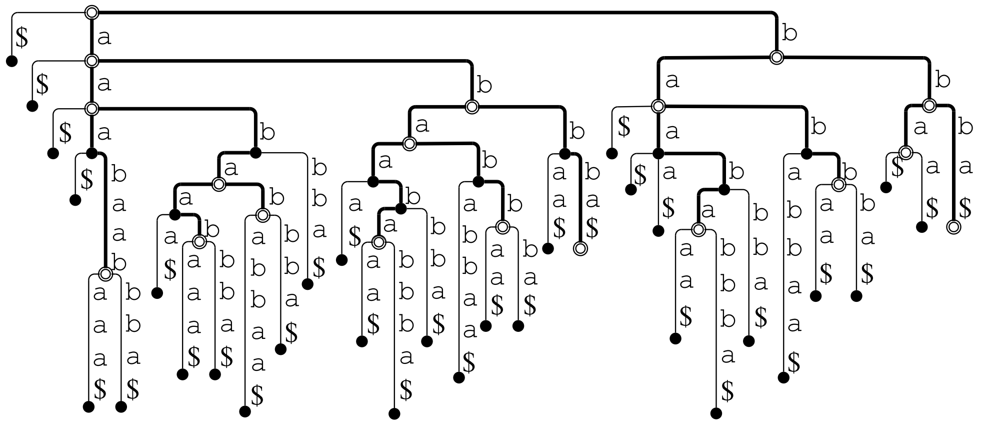


Figure 5.5: Illustration of $STree(D\$)$, where the double lined circles represent the 2-nodes.

For any node s of $STree(D\$)$, let $L(s)$ be the sequence of non-negative integers d arranged in the increasing order such that $d = 0$ or s is a d-node. Let A be the sequence obtained by concatenating $L(s)$ according to the pre-order of nodes s of $STree(D\$)$. Let u and v be the loci of x in $STree(D\$)$ and $STree(M_d)$, respectively. Then v corresponds to the leftmost occurrence of d in $A[i+1..]$ such that i is the position of j-th occurrence of 0 where j is the rank of u in the pre-order traversal of $STree(D\$)$. Thus v can be computed from u as follows. For the rightmost leaf l_u of the subtree rooted at u, $v = nil$ if $rank_A(d, select_A(0, PreOrd(u))) > select_A(0, PreOrd(l_u))$, and otherwise, v corresponds to $A[rank_A(d, select_A(0, PreOrd(u)))]$, where $PreOrd(s)$ denotes the rank of a node s in the pre-order traversal of $STree(D\$)$.

The numbers of 0's and d's in the array A are $O(n)$ and $O(n/d)$, respectively, and hence we have $|A| = O(n \log m)$. By Lemma 17, we can compute the locus of x in $STree(M_d)$ from the locus of x in $STree(D^R\$)$ in $O(\log\log m)$ time using $O(n \log m)$ space. □

Proof of Lemma 23

By Lemma 15, $xyz \in M_d$ implies that yz is d-right maximal. For each d-right maximal string β, let $len(\beta)$ denote the length of the shortest string α with $\alpha\beta \in M_d$. Then the desired string xy can be obtained from the d-right maximal extension yz of y that minimizes $len(yz)$.

From the locus of y^R in $STree(D^R\$)$, the locus of $(xy)^R$ in $STree(D^R\$)$ can be computed in three steps (see Figure 5.6).

Step 1: From the locus of y^R in $STree(D^R\$)$, find the locus of y in $STree(M_d)$.

Step 2: From the locus of y in $STree(M_d)$, find the locus of $(xyz)^R$ in $STree(D^R\$)$ such that x is one of the shortest strings x satisfying $xyz \in M_d$ for some string z.

Step 3: From the locus of $(xyz)^R$, find the locus of $(xy)^R$ in $STree(D^R\$)$.

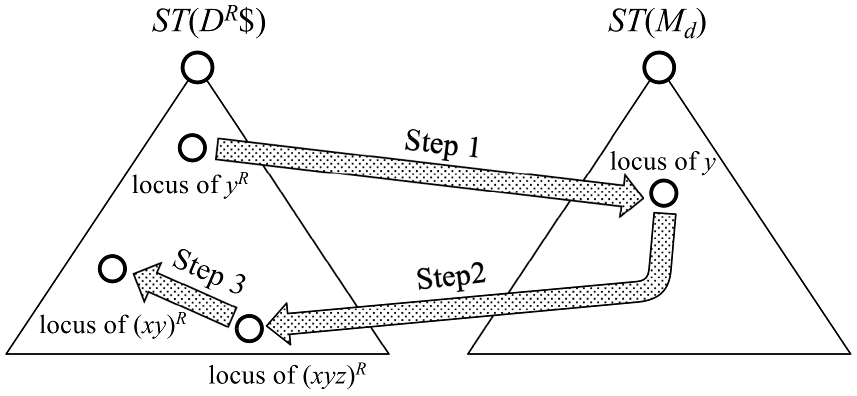


Figure 5.6: Computing the locus of $(xy)^R$ from the locus of y^R in $STree(D^R\$)$.

Step 1 requires $O(\log\log m)$ time by using the $O(n \log m)$-size data structure stated in Lemma 24.

For Step 2, we define two functions len and link on the set of nodes of $\mathit{STree}(M_d)$ as follows: For any node u of $\mathit{STree}(M_d)$, let β be the string represented by u. If there is some string α such that $\alpha\beta \in \mathit{Pre}(M_d)$, choose α as short as possible, and let $\mathit{len}(u) = |\alpha|$ and let $\mathit{link}(u)$ be the locus of α in $\mathit{STree}(D^R\$)$. If there is no such α, let $\mathit{len}(u) = \infty$ and $\mathit{link}(u) = \mathit{nil}$.

Suppose that v is the descendant of the locus of y in $\mathit{STree}(M_d)$ that minimizes $\mathit{len}(v)$. Then $\mathit{len}(v) = |x|$ and $\mathit{link}(v)$ is the locus of $(xyz)^R$ in $\mathit{STree}(D^R\$)$ since yz is d-right maximal. The locus of $(xyz)^R$ in $\mathit{STree}(D^R)$ can then be computed in constant time by storing the values $\mathit{len}(u)$ and $\mathit{link}(u)$ into the nodes u of $\mathit{STree}(M_d)$ and applying the Range Minimum Query technique.

In Step 3, the locus of $(xy)^R$ is obtained from the locus of $(xyz)^R$ in $\mathit{STree}(D^R\$)$ by traversing suffix links $|x|$ times. The task can be done in constant time by using the $O(n)$ space data structure for the level ancestor query [10] on suffix link tree of $\mathit{STree}(D^R\$)$.

Step 1 through Step 3 can be done in $O(\log\log m)$ time using $O(n \log m)$ space. □

5.5 Conclusion

In this chapter, we addressed the left-right maximal generic words problem and developed an $O(n \log m)$ size data structure, which answers queries in $O(|p| + o \log\log m)$ time, where o is the size of outputs. Our method is better than the previous work by Nishimoto et al. [67] both in the space requirement and in the query time. We achieved the $O(n \log m)$ space requirement by substituting $\mathit{STree}(M_d)$ for $\mathit{STree}(M_d\$)$, with the conjecture that the total size of $\mathit{STree}(M_d\$)$'s for $d = 1, \ldots, m$ are $\Theta(nm)$. To prove that the total size is $\Omega(nm)$ is left as future work.

Chapter 6

Finding Gapped Palindromes Online

A *palindrome* is a string of form xax^R, where x is a string called the left arm, a is either the empty string or a single character, and x^R is the reversed string of x called the right arm. Finding palindromic substrings in a given string w is a classical problem on string processing. The earliest work on this problem dates back to at least 1970's when Manacher [60] proposed an online algorithm to find all prefix palindromes in w in $O(n)$ time, where n is the length of w. Later, Apostolico et al. [3] pointed out that Manacher's algorithm can be used to find all maximal palindromes in w in $O(n)$ time, where a maximal palindrome is a substring palindrome $w[i..j] = w[i..j]^R$ of w whose arms cannot be further extended based on the same center $\frac{i+j}{2}$.

A natural generalization of palindromes is *gapped palindromes* of form xyx^R, where y is a string of length at least 2 called a *gap*[1]. Finding gapped palindromes has applications in bioinformatics, e.g.; RNA secondary structures called hairpins can be regarded as a kind of gapped palindrome $xy\overline{x}^R$, where $\overline{x}$ represents the complement of x ($\overline{x}$ is obtained by exchanging `A` with `U` and exchanging `C` with `G` in x). The most basic type of gapped palindromes is *g-gapped palindromes*, where $g \geq 2$ is a pre-defined fixed length of the gaps. For three parameters $g_{\min}$, $g_{\max}$, and A such that $2 \leq g_{\min} \leq g_{\max}$ and $A \geq 1$, Kolpakov and Kucherov [45] introduced *length-constrained gapped palindromes* (*LCGPs*) which has arms of length at least A and gaps of length in range $[g_{\min}, g_{\max}]$. This is a natural generalization of g-gapped palindromes with $g_{\min} = g_{\max} = g$ and $A = 1$.

In this chapter, we consider the problems of finding these gapped palindromes in a string in an *online manner*. Namely, our input is a growing string to which new characters can be appended, and each character of the string arrives one by one, from left to right. Let n be the

[1]If y is a single character, then xyx^R is a palindrome of odd length. Thus we here assume y is of length at least 2.

length of the final string w. We propose:

(1) An online algorithm to compute all maximal g-gapped palindromes in w in $O(n \log \sigma)$ time and $O(n)$ space, where σ is the alphabet size. This algorithm can be modified to output only *distinct* maximal g-gapped palindromes in an online manner, in the same complexity.

(2) An online algorithm to compute all maximal LCGPs in w in $O(n(M + \log \sigma))$ time and $O(n)$ space, where $M = \max\{\frac{g_{\max} - g_{\min}}{A}, 1\}$.

Formal definitions of the maximality of these gapped palindromes will be given in Section 6.2 and Section 6.3, respectively.

We remark that using a slightly modified version of Solution (1), it is trivial to obtain an $O(n(g_{\max} - g_{\min} + \log \sigma))$-time solution for finding all maximal LCGPs, by simply testing gap lengths $g_{\min}, g_{\min} + 1, \ldots, g_{\max}$ separately. Hence, in the case where A is not a constant and $\log \sigma$ is not a dominating term, then Solution (2) speeds up this trivial method by a factor of A. On the other hand, in the case where A is a constant, then we show that there exists a string of length n which contains $\Omega(nM)$ maximal LCGPs, meaning that we cannot hope significant speed-up in the worst case.

Solution (2) is based on Solution (1) and is quite different from the offline solution by Kolpakov and Kucherov [45]. To our knowledge, these are the first efficient online algorithms that compute *any kind* of gapped palindromes.

Related work. A number of efficient *offline* algorithms for computing various kinds of gapped palindromes have been proposed in the literature.

Let w be an input string w of length n over the integer alphabet. There exists a folklore $O(n)$-time algorithm (see e.g. [39]) which finds all maximal g-gapped palindromes for a given fixed gap length g; the Ukkonen tree [75, 28] of string $w^R\#w\$$ and a constant-time LCA data structure [9] over the Ukkonen tree are constructed during preprocessing, and then computing each maximal g-gapped palindrome reduces to an outward longest common extension (LCE) query, which can be answered by an LCA query on the tree. Our algorithm for computing all maximal g-gapped palindromes can be regarded as an online version of this algorithm.

Kolpakov and Kucherov [45] proposed an $O(n + L)$-time offline algorithm to find all maximal LCGPs, where L is the number of outputs. Their algorithm consists of the following two steps: In the first step, it computes all (not necessarily outward maximal) LCGPs whose arms are of length exactly A. Let (i, j) be the pair of the ending position i and the beginning position j of the left and right arms of each of the above LCGPs, respectively. In the second step, for each LCGP computed above, the algorithm performs an outward LCE query from i and j, using the same suffix-tree based data structure as for the maximal g-gapped palindromes above. However, each time a new character is appended to the growing string, the LCE value from the same pair of positions may increase, and it is impossible to know beforehand when the growth of the LCE value for each pair of positions stops. Thus, it seems difficult to apply Kolpakov and Kucherov's solution to our online setting.

There exist efficient offline solutions for finding other kinds of gapped palindromes. Kolpakov and Kucherov [45] also proposed an $O(n)$-time[2] offline algorithm to compute all maximal *long-armed palindromes* (those whose arms are longer than their gap) in a given string w of length n. Kolpakov and Kucherov's algorithm uses a variant of Lempel-Ziv factorization called the reversed LZ factorization of strings. Let $f_1, \dots, f_k$ be the reversed LZ factorization of w. Then, for each pair f_i of adjacent factors, their algorithm focuses on positions $\frac{|f_i|}{2^k}$ for every $1 \le k \le \lceil \frac{|f_i|}{2} \rceil$ in f_i. This implies that the length of each f_i needs to be precomputed. However, in the online setting, the length of the last factor that is a suffix of the current string can extend each time a new character is appended. It is therefore unclear whether we can extend their solution to the online scenario.

Gawrychowski et al. [33] considered a generalization of long-armed palindromes called *ν-gapped palindromes*; For a parameter $\nu > 1$, a gapped palindrome xyx^R is said to be a ν-gapped palindrome iff $|xy| \le \nu|y|$. Gawrychowski et al. [33] proposed an $O(\nu n)$-time offline algorithm which computes all maximal ν-gapped palindromes in an input string w of length n. This algorithm requires a preprocessing of the input w for integer $d \ge 2$ such that the occurrences of a substring of length 2^k (called a basic factor therein) in another substring of length $d2^k$ can be computed efficiently. Thus, it seems difficult to apply their result to the online setting. After, Gawrychowski et al. [34] proposed an optimal $O(\nu n)$ time algorithm for computing ν-gapped palindromes for integer alphabets.

[2]Originally, Kolpakov and Kucherov [45] stated their algorithm works in $O(n+S)$ time, where S is the number of outputs. It follows from a recent work by Gawrychowski et al. [33] that $S = O(n)$.

6.1 Notations and Tools

6.1.1 Gapped palindromes

A string p is said to be a *gapped palindrome* iff $p = xyx^R$ for some non-empty strings x, y with $|y| > 1$. The intervals $[1, |x|]$, $[|y| + 1, |xy|]$, and $[|xy| + 1, |xyx|]$ in p are called the *left arm*, *gap*, and *right arm* of gapped palindrome $p = xyx^R$. Note that in general the choice of arms and gap are not unique for the same string p. For instance, if $p = \texttt{abccbba}$, then we can take $x = \texttt{ab}$ and $y = \texttt{ccb}$, or $x = \texttt{a}$ and $y = \texttt{bccbb}$.

A gapped palindrome xyx^R is said to be a *length-constrained palindrome* (*LCGP*) iff $|x| \geq A$ and $g_{\min} \leq |y| \leq g_{\max}$ for some fixed integer parameters $A \geq 1$ and $1 < g_{\min} \leq g_{\max}$. A gapped palindrome xyx^R is said to be a g-*gapped* palindrome iff $|y| = g$ for some fixed integer $g > 1$. Note that any g-gapped palindrome is a special case of a length-constrained palindrome with $g_{\min} = g_{\max} = g$ and $A = 1$.

An occurrence of a gapped palindrome $p = xyx^R$ in a string w is identified by a triple (i, j, a) such that a denotes the length of each arm, and i, j denote the ending and beginning positions of the left and right arms of p, respectively. Namely, $w[i-a+1..i] = x$, $w[i+1..j-1] = y$, and $w[j..j + a - 1] = x^R$. The *center* of an occurrence (i, j, a) of a gapped palindrome in w is $\frac{i+j}{2}$.

6.1.2 LCE queries

Given an ordered pair (i, j) of positions in a string w of length n, a *reversed longest common extension query* $RLCE_w(i, j)$ returns $LCE((w[1..i])^R, w[j..n])$. Computing $RLCE_w(i, j)$ reduces to the lowest common ancestor (LCA) problem on $UTree(w')$, where $w' = w^R\#w$ and $\#$ is a special delimiter which does not occur in w. Let $v_{i,j}$ be the LCA of the two leaves which represent the suffixes $w'[n - i + 1..2n + 1]$ and $w'[n + j + 1..2n + 1]$. Then, we have that $|str(v_{i,j})| = RLCE_w(i, j)$. Using an LCA data structure (e.g. [9]), we can answer $RLCE_w(i, j)$ query for any pair (i, j) of positions in $O(1)$ time after an $O(n)$-time preprocessing on $UTree(w')$.

6.2 Online Algorithms to Compute All Maximal g-gapped Palindromes

An occurrence (i, j, a) of a g-gapped palindrome xyx^R in a string w is said to be *maximal*, if the arms x, x^R cannot be extended outward, i.e., if $w[b-1] \neq w[e+1]$, $b = 1$, or $e = n$, where $b = i - a + 1$ and $e = j + a - 1$[3].

Example 7. *Consider string* `aabaacabbcaabb` *and let* $g = 3$*. This string has* 3*-gapped maximal palindromes* $(1, 5, 1) = \mathtt{a} \cdot \mathtt{aba} \cdot \mathtt{a}$*,* $(6, 10, 4) = \mathtt{baac} \cdot \mathtt{ab\,b} \cdot \mathtt{caab}$*,* $(7, 11, 1) = \mathtt{a} \cdot \mathtt{bbc} \cdot \mathtt{a}$*, and* $(9, 13, 2) = \mathtt{bb} \cdot \mathtt{caa} \cdot \mathtt{bb}$.

6.2.1 Computing all maximal g-gapped palindromes online

In this subsection, we propose online algorithms to compute all maximal g-gapped palindromes in a string w of length n, where $g > 1$ is a given fixed integer parameter (since $g = 1$ gives odd palindromes, we set $g > 1$).

As was mentioned in the begging of Chapter 6, there exists an *offline* algorithm which computes all g-gapped maximal palindromes in $O(n)$ time and space for an input string w of length n over an integer alphabet. However, in our scenario the input string w is given online, and we wish to process each character from left to right. In the sequel, we will show our online algorithm which can deal with this setting.

For each $k = 1, \ldots, n$, our algorithm maintains the *longest* g-gapped suffix palindrome of $w[1..k]$ (if it exists). For each g-gapped palindrome to compute, we maintain two variables i, j ($i < j < k$) that represent the ending position of the left arm and the beginning position of the right arm of g-gapped palindrome, respectively. Assume $(i, j, a_{i,j})$ is the longest g-gapped suffix palindrome of $w[1..k]$, where the gap of length g is $w[i+1..j-1]$, $j = i + g + 1$ and $j + a_{i,j} - 1 = k$. In case there are no g-gapped suffix palindromes of $w[1..k]$, then let $a_{i,j} = 0$, $i = k - g$ and $j = k + 1$. Depending on the next character $w[k+1]$, we have two cases:

1. If $w[i - a_{i,j}] = w[k+1]$, then there exists a longer g-gapped palindrome centered at $\frac{i+j}{2}$. We then naïvely extend the arm length by $a_{i,j} \leftarrow a_{i,j} + 1$, and proceed to the forthcoming character by updating $k \leftarrow k + 1$.

[3]Since the gap length is fixed to g and since it simplifies the description of the algorithm, here we do not consider inward maximality of the arms. However, it is easy to modify our algorithm so that it outputs all g-gapped palindromes that are both outward and inward maximal with the same efficiency.

2. If $w[i - a_{i,j}] \neq w[k + 1]$, then it appears that $(i, j, a_{i,j})$ is the longest g-gapped maximal palindrome ending at position k, and hence we output it. We then shift the gap to the right by updating $i \leftarrow i + 1$ and $j \leftarrow j + 1$. There are two-sub cases.
   (a) If $j > k+1$, then it appears that there is no g-gapped suffix palindrome of $w[1..k+1]$. We therefore update $k \leftarrow k + 1$ and proceed to the forthcoming character, with the current values of i and j.
   (b) If $j \leq k+1$, then we compute $a_{i,j}$ (we will later describe how to efficiently compute it for updated i and j). There are two sub-cases:
      i. If $j + a_{i,j} - 1 = k + 1$, then $(i, j, a_{i,j})$ is the longest g-gapped suffix palindrome of $w[1..k+1]$. We proceed to the forthcoming character by updating $k \leftarrow k+1$.
      ii. If $j + a_{i,j} - 1 < k + 1$, then $(i, j, a_{i,j})$ is the maximal g-gapped palindrome with the gap beginning at position $i + 1$, and hence we output it. We then shift the gap to the right by updating $i \leftarrow i + 1$ and $j \leftarrow j + 1$, and go to either Case 2a or Case 2b depending on the value of j.

In order to efficiently compute $a_{i,j}$ of Case 2 above in our online scenario, we utilize the following results:

Theorem 13 ([40]). *There exists an $O(n \log \sigma)$-time $O(n)$-space algorithm to maintain the Ukkonen tree with suffix links for a bidirectionally growing string to which new characters can be prepended and appended, where n is the length of the final string.*

Theorem 14 ([18]). *There exists a linear-space algorithm for a rooted tree that supports the following operations and query in $O(1)$ worst-case time: which supports the following operations and query in $O(1)$ worst-case time: (1) Insert a new node; (2) Delete an existing node; (3) LCA query for any pair of nodes in the current tree.*

We are ready to show the main result of this section:

Theorem 15. *For a growing string to which new characters are appended, we can compute all maximal g-gapped palindromes in an online manner, in $O(n \log \sigma)$ time and $O(n)$ space, where n is the length of the final string.*

Proof. The correctness immediately follows from the above arguments.

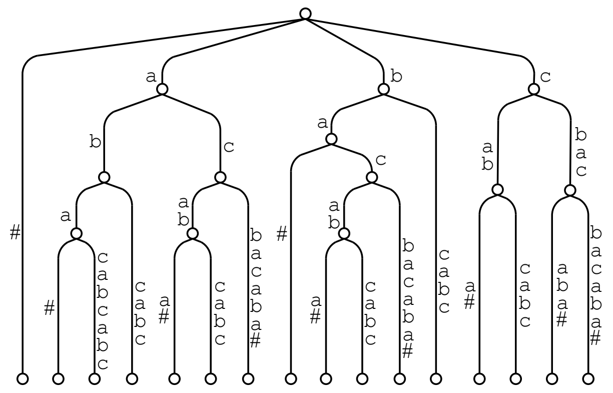


Figure 6.1: $\mathit{UTree}(w'_k)$ with $w[1..k]$ = abacabcabc and w'_k = cbacbacaba#abacabcabc. The label strings after # are omitted for simplicity.

The time complexity is shown as follows. In the sequel, we consider the amortized time cost for each $k = 1, \ldots, n$. For each k that falls into Case 1, it clearly takes $O(1)$ time. For each k that falls into Case 2b, we output several maximal g-gapped palindromes. It takes $O(1)$ time to output the longest maximal g-gapped palindrome. The key is how to compute the arm lengths $a_{i,j}$ of shorter maximal g-gapped palindromes. For this sake we maintain $\mathit{UTree}(w'_k)$ where $w'_k = (w[1..k])^R\#w[1..k]$, where # is a special delimiter which does not appear elsewhere in w'_k (see also Fig. 6.1 for an example).

Note that computing $a_{i,j}$ is equivalent to computing $\mathit{RLCE}_{w[1..k]}(i, j)$, and thus is equivalent to computing $|\mathit{str}(v_{i,j})|$, where $v_{i,j}$ is the LCA of the nodes of $\mathit{UTree}(w'_k)$ which represent the suffixes $w'_k[k-i+1..2k+1]$ and $w'_k[k+j+1..2k+1]$ of w'_k. Since # is unique in w'_k, the suffix $w'_k[k-i+1..2k+1]$ is always represented by a leaf of $\mathit{UTree}(w'_k)$ and hence can easily be accessed in $O(1)$ time. However, notice that the other suffix $w'_k[k+j+1..2k+1]$ is not represented by a node when the path that spells out $w'_k[k+j+1..2k+1]$ from the root ends on an edge (this can happen when $w'_k[k+j+1..2k+1] = w[j..k]$ is a prefix of another suffix of $w[1..k]$). Consider such a case, and let $\langle u_j, s_j, t_j\rangle$ be the locus for the suffix $w'_k[k+j+1..2k+1]$. Since u_j is the nearest ancestor to the locus, we can use u_j for the LCA query instead of the locus for $w'_k[k+j+1..2k+1]$.

What remains is how to quickly find the loci for increasing j. For this we can use a similar technique to Ukkonen's online Ukkonen tree construction algorithm [74]: Assume that the locus $\langle u_j, s_j, t_j\rangle$ for the suffix $w'_k[k+j+1..2k+1] = w[j..k]$ in $\mathit{UTree}(w'_k)$ is given. To find the locus for $\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$ for the next suffix $w'_k[k+j+2..2k+1] = w[j+1..k]$, we first follow the suffix link of u_j and arrive at $z = \mathit{slink}(u_j)$. We then traverse the path from z which spells out

$w'_k[s_{j+1}..t_{j+1}]$. The last piece of this path gives the locus $\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$ (see also Fig. 6.2).

Using a similar analysis to [74], the cost to find this locus is amortized to $O(\log\sigma)$. Since the total number of outputs (maximal g-gapped palindromes) is linear in n, the amortized cost per output is $O(\log\sigma)$. The cost to update $\mathit{UTree}(w'_k)$ to $\mathit{UTree}(w'_{k+1})$ is amortized to $O(\log\sigma)$ by Theorem 13. Each LCA query can be answered in $O(1)$ time by Theorem 14. Hence, the total time complexity is $O(n\log\sigma)$. The total space requirement is clearly $O(n)$. This completes the proof. □

6.2.2 Computing all distinct maximal g-gapped palindromes online

Consider a g-gapped palindrome $p = xyx^R$ which has at least two maximal occurrences in a string w. When considering "distinctness" of two maximal occurrences (i, j, a) and (i', j', a) of p, we take into account the left and right neighbouring characters for a technical reason. Namely, two maximal occurrences (i, j, a) and (i', j', a) of a g-gapped palindromes are said to be *distinct* iff (1) $w[b-1] \neq w[b'-1]$ or (2) $w[e+1] \neq w[e'+1]$, where $b = i-a+1$, $e = j+a-1$, $b' = i'-a+1$, and $e' = j'+a-1$.

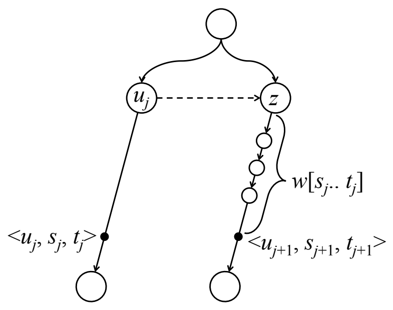

u_j
z
$w[s_j..t_j]$
$\langle u_j, s_j, t_j\rangle$
$\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$

Figure 6.2: Illustration of how to find the locus $\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$ of the next suffix $w'_k[k+j+2..2k+1] = w[j+1..k]$ using the suffix link of u_j, where $\langle u_j, s_j, t_j\rangle$ is the locus of the previous suffix $w'_k[k+j+1..2k+1] = w[j..k]$. The cost for walking down from node z to the locus for $\langle u_{j+1}, s_{j+1}, t_{j+1}\rangle$ is $O(\log\sigma)$ amortized.

Our online algorithm of Section 6.2.1 can be modified to output all distinct maximal g-gapped palindromes in an online manner.

For any string w, let $\mathit{lusuf}(w)$ denote the longest suffix of w which appears at least twice in w (we assume that the empty string ε appears $|w|+1$ times in w so $\mathit{lusuf}(w)$ always exists). We make use of the following simple observation:

Observation 1. *Let (i, j, a) be an occurrence of a maximal g-gapped palindrome xyx^R in a string w, and let $c_\ell = w[i-a]$ and $c_r = w[j+a]$. Then, it is the first (i.e. left-most) maximal occurrence of xyx^R in w iff $|c_\ell xyx^R c_r| = j-i+2a+1 > |\mathit{lusuf}(w[1..j+a-1])|$.*

Theorem 16. *For a growing string to which new characters are appended, we can compute all distinct maximal g-gapped palindromes in an online manner, in $O(n\log\sigma)$ time and $O(n)$*

space, where n is the length of the final string.

Proof. On top of $\mathit{UTree}(w'_k)$ used in Theorem 15, we build another Ukkonen tree $\mathit{UTree}(w[1..k])$ for increasing $k = 1, \ldots, n$ using Ukkonen's online algorithm [74]. For each k, Ukkonen's algorithm maintains an invariant called the *active point* which indicates the locus of $\mathit{lusuf}(w[1..k])$. When we process the kth character $w[k]$, we store $|\mathit{lusuf}(w[1..h])|$ for all $1 \leq h \leq k$. Let $(i, j, a_{i,j})$ be an occurrence of a maximal g-maximal found at the k-th stage of the algorithm where we have processed $w[1..k]$. Then, we can determine in $O(1)$ time whether or not it is the first maximal occurrence of the g-gapped palindrome using Observation 1 (recall that the right mismatched position $j + a_{i,j}$ never exceeds k and hence we know $|\mathit{lusuf}(w[1..j + a_{i,j}])|$). Since Ukkonen's online algorithm works in $O(n \log \sigma)$ time and $O(n)$ space, the theorem holds. □

We note that a similar technique was used by Kosolobov et al. [49] in their online algorithm to find all distinct palindromes (without gaps) in a given string.

6.3 Online Algorithms to Compute All Maximal LCGPs

An occurrence (i, j, a) of an LCGP in a string w of length n is said to be *outward-maximal* iff $w[i - a] \neq w[j + a]$, $i - a + 1 = 1$, or $j + a - 1 = n$, and it is said to be *inward-maximal* iff $w[i+1] \neq w[j-1]$. It is said to be *maximal* iff it is both outward-maximal and inward-maximal[4].

Example 8. *Consider string* `aabaacabbcaabb` *and let* $g_{\min} = 1$, $g_{\max} = 4$, *and* $A = 2$. *All the maximal LCGPs in this string are* $(2, 4, 2) = \texttt{aa} \cdot \texttt{b} \cdot \texttt{aa}$, $(4, 7, 2) = \texttt{ba} \cdot \texttt{ac} \cdot \texttt{ab}$, $(6, 10, 4) = \texttt{baac} \cdot \texttt{abb} \cdot \texttt{caab}$, *and* $(9, 13, 2) = \texttt{bb} \cdot \texttt{caa} \cdot \texttt{bb}$.

6.3.1 Computing all maximal LCGPs online

In this section, we present an online algorithm to compute all maximal LCGPs of a given string w. This algorithm works in $O(n(M + \log \sigma))$ time and $O(n)$ space, where $n = |w|$ and $M = \max\{\frac{g_{\max} - g_{\min}}{A}, 1\}$.

Let $d = \frac{g_{\max} - g_{\min}}{2}$. For ease of explanation, we assume that $d \bmod A = 0$ and we will describe our algorithm for this case. However, the algorithm can easily be extended to a general case with $d \bmod A \neq 0$, retaining the same efficiency.

[4]Since the gap length varies in range $[g_{\min}, g_{\max}]$, we here consider both outward and inward maximality of the arms.

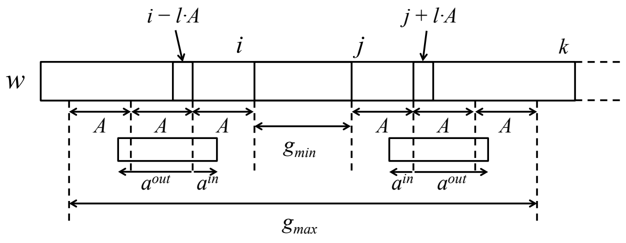


Figure 6.3: Illustration for Lemma 25. Since any LCGP centered at $\frac{i+j}{2}$ with gap length in range $[g_{\min}, g_{\max}]$ contains a pair $(i - l \cdot A, j + l \cdot A)$ of sampled positions for some l, we can compute it by two LCEs from the sampled positions.

For each $k = 1, \ldots, n$ in increasing order, we maintain a pair (i, j) of positions such that $j - i = g_{\min} + 1$ and the longest inward-maximal suffix LCGP of $w[1..k]$ is centered at $\frac{i+j}{2}$ (if it exists). If it does not exist, then let $i = k - g_{\max}$ and $j = k - g_{\max} + g_{\min} + 1$. For $1 \leq l \leq \frac{d}{A}$, we consider the positions $i - l \cdot A$ and $j + l \cdot A$ in $w[1..k]$, called *sampled* positions. The following simple lemma suggests how we can use these sampled positions for efficient computation of LCGPs.

Lemma 25. *Let (i', j', a') be any maximal LCGP whose center is $\frac{i+j}{2}$ (i.e., $\frac{i'+j'}{2} = \frac{i+j}{2}$). Then, there exists l ($1 \leq l \leq \frac{d}{A}$) such that $j+l{\cdot}A \in [j', j'+a'-1]$ and $i-l{\cdot}A \in [i'-a'+1, i']$. Moreover, for each such l, (i', j', a') is the unique maximal LCGP satisfying the above conditions.*

Proof. The existence of l is clear from the fact that the arms of LCGPs must be at least A long (see also Figure 6.3). By definition, the arms of two different maximal LCGPs with the same center cannot overlap. Thus, for each l, there exists at most one LCGP whose left and right arms contain sampled positions $i - l \cdot A$ and $j + l \cdot A$, respectively. This completes the proof. □

Let l ($1 \leq l \leq \frac{d}{A}$) be the smallest integer such that $i - l \cdot A$ (resp. $j + l \cdot A$) is contained in the left arm (resp. the right arm) of the longest suffix inward-maximal LCGP of $w[1..k]$ that is centered at $\frac{i+j}{2}$, and let a_l be the length of the arm of this LCGP. Also, let i_l, j_l be the ending position of the left arm and the beginning position of the right arm of this LCGP, respectively. Note $\frac{i_l+j_l}{2} = \frac{i+j}{2}$ and $j_l + a_l - 1 = k$. Depending on the next character $w[k + 1]$, we have two cases:

1. If $w[i_l - a_l] = w[k + 1]$, then $(i_l, j_l, a_l + 1)$ is the longest suffix inward-maximal LCGP of $w[1..k + 1]$ centered at $\frac{i+j}{2}$. Thus, we naïvely extend the arm length outward by $a_l \leftarrow a_l + 1$, and proceed to the forthcoming character by updating $k \leftarrow k + 1$.

2. If $w[i_l - a_l] \neq w[k+1]$, then it appears that (i_l, j_l, a_l) is a maximal LCGP centered at $\frac{i+j}{2}$ and ending at position k, and hence we output it. To compute other maximal LCGPs centered at $\frac{i+j}{2}$, we do the following: We update $l \leftarrow l+1$, and consider a pair $(i - l \cdot A, j + l \cdot A)$ of the sampled positions and compute the outward LCE $a_l^{out} = RLCE_{w[1..k+1]}(i - l \cdot A, j + l \cdot A)$ and the inward LCE $a_l^{\text{in}} = RLCE_{w[1..k+1]}(j + l \cdot A - 1, i - l \cdot A + 1)$ from these sampled positions (see also Fig. 6.3). There are three sub-cases depending on the LCE values:
   (a) If $a_l^{out} + a_l^{\text{in}} < A$ or $a_l^{\text{in}} > l \cdot A$, then there is no maximal LCGP with gap length in range $[g_{\min}, g_{\max}]$ that is centered at $\frac{i+j}{2}$ and contains the sampled positions $i - l \cdot A$ and $j + l \cdot A$. We update $l \leftarrow l+1$, and go to one of the following sub-cases.
      i. If $l \leq \frac{d}{A}$, then we compute the outward and inward LCEs from the pair of sampled positions with l.
      ii. If $l > \frac{d}{A}$, then there is no suffix gapped palindrome of $w[1..k]$ that is centered at $\frac{i+j}{2}$ and has a gap length in range $[g_{\min}, g_{\max}]$. We therefore update $i \leftarrow i+1$, $j \leftarrow j+1$, $l \leftarrow 1$, $k \leftarrow k+1$ and proceed to the forthcoming character.
   (b) If $a_l^{out} + a_l^{\text{in}} \geq A$, $a_l^{\text{in}} \leq l \cdot A$, and $j + l \cdot A + a_l^{out} \leq k$, then (i_l, j'_l, a_l) is a maximal LCGP centered at $\frac{i+j}{2}$ where $i_l = i - l \cdot A + a_l^{\text{in}}$, $j + l \cdot A + a_l^{out}$, and $a_l = a_l^{out} + a_l^{\text{in}}$. We output it and update $l \leftarrow l + 1 + \lfloor \frac{a_l^{out}}{A} \rfloor$ (this is to skip the subsequent sampled positions which are also contained in the same LCGP due to Lemma 25).
      i. If $l \leq \frac{d}{A}$, then we compute the outward and inward LCEs from the pair of sampled positions with l.
      ii. If $l > \frac{d}{A}$, then there is no inward-maximal suffix gapped palindrome of $w[1..k]$ that is centered at $\frac{i+j}{2}$ and has a gap length in range $[g_{\min}, g_{\max}]$. We therefore update $i \leftarrow i+1$, $j \leftarrow j+1$, $l \leftarrow 1$, $k \leftarrow k+1$ and proceed to the forthcoming character.
   (c) If $a_l^{out} + a_l^{\text{in}} \geq A$, $a_l^{\text{in}} \leq l \cdot A$, and $j + l \cdot A + a_l^{out} = k+1$, then (i_l, j_l, a_l) is an inward-maximal gapped suffix palindrome of $w[1..k+1]$ with gap length in range $[g_{\min}, g_{\max}]$. Moreover, since we have processed l in increasing order, it is guaranteed that (i_l, j_l, a_l) is the longest such one. Hence, we proceed to the next character by updating $k \leftarrow k+1$.

Theorem 17. *For a growing string to which new characters are appended, we can compute all LCGPs in an online manner, in $O(n(M + \log \sigma))$ time and $O(n)$ space, where n is the length of the final string and $M = \max\{\frac{g_{\max} - g_{\min}}{A}, 1\}$.*

Proof. The correctness should be clear from the above arguments.

For each $k = 1, \ldots, n$, we consider a fixed center $\frac{i+j}{2}$ and compute all LCGPs with this center. We perform at most $\frac{2d}{A}$ LCE queries for each k, as there are $\frac{d}{A}$ sampled positions for each k. Since each LCE query can be answered in $O(1)$ time as in the proof of Theorem 15, the total time cost of the LCE queries for all $k = 1, \ldots, n$ is $O(\frac{d}{A}n) = O(Mn)$. We use additional $O(n \log \sigma)$ time to maintain the Ukkonen tree augmented with the dynamic LCA data structure for bidirectionally growing string $w'_k = (w[1..k])^R \# w[1..k]$. Thus the total time complexity is $O(n(M + \log \sigma))$.

The total space requirement is dominated by the Ukkonen tree and the dynamic LCA data structure, and hence is $O(n)$. □

6.3.2 Optimality of our algorithm

The following corollary is immediate from Theorem 17.

Corollary 1. *For constant parameters $g_{\min}$, $g_{\max}$, A and a constant-size alphabet, we can compute all maximal LCGPs in a string of length n in an online manner, in optimal $O(n)$ time and space.*

We can show that even for non-constant gap constraints $g_{\min}$ and $g_{\max}$, the running-time of our algorithm is optimal in the worst case. For any string w, let L_w denote the number of all maximal LCGPs in w w.r.t. given parameters $g_{\min}$, $g_{\max}$, and A. It immediately follows from Lemma 25 that L_w is upper-bounded by the total number of sampled positions in w. Hence $L_w = O(Mn)$, where $n = |w|$ and $M = \max\{\frac{g_{\max} - g_{\min}}{A}, 1\}$. It is also true that there is an instance w for which $L_w = \Omega(Mn)$ if A is a constant: For example, consider string $z = (\mathtt{abc})^{\frac{n}{3}}$. This string z contains maximal gapped palindromes of form $\mathtt{a}(\mathtt{bc}(\mathtt{abc})^p)\mathtt{a}$ with arm $\mathtt{a}$, $\mathtt{b}(\mathtt{c}(\mathtt{abc})^p\mathtt{a})\mathtt{b}$ with arm $\mathtt{b}$, and $\mathtt{c}((\mathtt{abc})^p\mathtt{ab})\mathtt{c}$ with arm $\mathtt{c}$ for all $0 \leq p \leq \frac{n}{3} - 2$. Thus, for $A = 1$ and for any $2 \leq g_{\min} \leq g_{\max}$, the string z contains $L_z = \Theta((g_{\max} - g_{\min})n) = \Theta(\frac{g_{\max} - g_{\min}}{A}n) = \Theta(Mn)$ maximal LCGPs. Hence the running time $O(M(n + \log \sigma))$ of our algorithm is optimal in the worst case, for a constant-size alphabet.

6.4 Conclusions

In this chapter, we presented an online algorithm which finds all maximal length-constrained gapped palindromes (LCGPs) occurring in a string w of length n in in $O(n(\frac{g_{\min}-g_{\max}}{A} + \log\sigma))$ time, for given parameters $2 \leq g_{\min} \leq g_{\max}$ and $A \geq 1$, where σ is the alphabet size. We also showed that if A is a constant, then there exists a string which contains $\Omega((g_{\min} - g_{\max})n)$ maximal LCGPs. This implies that for a constant-size alphabet the running time of our algorithm is optimal in the worst case.

To our knowledge, the proposed methods are the first online algorithms to find any kind of gapped palindromes in strings. Therefore, there remain many open problems. In particular, we are interested in the following:

- Is there a string of length n which contains $\Omega(\frac{g_{\min}-g_{\max}}{A}n)$ maximal LCGPs for *non-constant* A?
- Can we reduce the $n\frac{g_{\min}-g_{\max}}{A}$ factor to L_w in the $O(n(\frac{g_{\min}-g_{\max}}{A} + \log\sigma))$-time algorithm for finding all maximal LCGPs, thereby obtaining an optimal algorithm?
- Can the *maximal ν-gapped palindromes* [33] of a given string be computed online efficiently?

Chapter 7

Almost Linear Time Computation of Maximal Repetitions in Run Length Encoded Strings

Periodicity and repetitions in strings are one of the most important characteristic features in strings. They have been one of the first objects of study in the field of combinatorics on words [73] and have many theoretical, as well as practical applications, e.g., in bioinformatics [44].

Maximal repetitions are periodically maximal substrings of a string where the smallest period is at most half the length of the substring, i.e., there are at least two consecutive occurrences of the same substring. An $O(n \log n)$ time algorithm for computing all of the maximal repetitions contained in a string of length n, was shown by Main and Lorentz [59], which is optimal for general unordered alphabets, i.e., when only equality comparisons between the letters are allowed. Kolpakov and Kucherov [46] further showed that the number of maximal repetitions is actually $O(n)$, and gave a linear time algorithm for ordered constant size alphabets (and essentially for integer alphabets), to compute all of them. The algorithm was a modification of the algorithm by Main [58], which in turn relies on the Lempel-Ziv 77 (LZ77) factorization [77] of the string, which can be computed in linear time for ordered constant size or integer alphabets [21], but requires $\Omega(n \log \sigma)$ time for general ordered alphabets [47], where σ is the size of the alphabet. Recently, a new characterization of maximal repetitions using Lyndon words was proposed by Bannai et al. [6, 7], which lead to a very simple proof to what was known as the “runs” conjecture, i.e., that the number of maximal repetitions in a given string of length n is less than n [7]. The characterization also lead to a new linear time algorithm for computing maximal repetitions on ordered constant size and integer alphabets, which does not require the LZ77 factorization, but only on a linear number of *longest common extension queries*. Furthermore, based on this algorithm, the running time for computing all maximal repetitions for general ordered alphabets were subsequently improved to $O(n \log^{2/3} n)$ by Kosolobov [48], $O(n \log \log n)$ by Gawrychowski et al. [35], and $O(n\mathcal{A}(n))$ by Crochemore et al. [22], where $\mathcal{A}$ denotes the inverse Ackermann function.

In this chapter, we consider the problem of computing all maximal repetitions contained in a string when given the *run-length encoding* (RLE) of the string, which is a well known compressed representation where each maximal substring of the same character is encoded as a pair consisting of the letter and the length of the substring. For example, the run-length encoding of the string aaaabbbaaacc is $(\mathtt{a}, 4)(\mathtt{b}, 3)(\mathtt{a}, 3)(\mathtt{c}, 2)$. The main contributions of the chapter are:

1. an upper bound $m + k - 1$ on the number of maximal repetitions contained in a string, where m is the size of its run-length encoding and k is the number of run-length factors whose exponent is at least 2, and
2. an $O(m\mathcal{A}(m))$ time and $O(m)$ space algorithm to compute all maximal repetitions in a string.

Our algorithm is at least as efficient as the non-RLE algorithms for general ordered alphabets. Furthermore, when the input string is compressible via RLE, our algorithm can be faster and more space efficient compared to the non-RLE algorithms. Although our algorithm mimics those for non-RLE strings and is conceptually simple, its correctness is based on new non-trivial observations on the occurrence of specific Lyndon words in run-length encoded strings.

Efficient algorithms for string problems when the input is given in RLE has been considered in various contexts, for example, edit distance [15], various Longest Common Subsequence problems [54, 50], palindrome retrieval [16], computing Lempel Ziv factorization [76], etc. We shall repeat below a claim made in [50] concerning the significance of RLE-based solutions:

> “A common criticism against RLE based solutions is a claim that, although they are theoretically interesting, since most strings “in the real world” are not compressible by RLE, their applicability is limited and they are only useful in extreme artificial cases. We believe that this is not entirely true. There can be cases where RLE is a natural encoding of the data, for example, in music, a melody can be

expressed as a string of pitches and their duration. Furthermore, in the data mining community, there exist popular preprocessing schemes for analyzing various types of time series data, which convert the time series to strings over a fairly small alphabet as an approximation of the original data, after which various analyses are conducted (e.g. SAX (Symbolic Aggregate approXimation) [52], *clipped* bit representation [5], etc.). These conversions are likely to produce strings which are compressible by RLE (and in fact, shown to be effective in [5]), indicating that RLE based solutions may have a wider range of application than commonly perceived."

7.1 Definitions and Notations

In this chapter, we assume a general ordered alphabet, where a total order $\prec$ is defined on Σ, and the order between two letters in the alphabet can be computed in constant time. A total order $\prec$ on the alphabet induces a total order on the set of strings called the *lexicographic order*, which we also denote by $\prec$, i.e., for any $x, y \in \Sigma^*$, $x \prec y \iff x$ is a proper prefix of y, or, there exists $1 \leq i \leq \min\{|x|, |y|\}$ s.t. $x[1..i-1] = y[1..i-1]$ and $x[i] \prec y[i]$.

All previous linear time algorithms either assume a constant size ordered alphabet or an integer alphabet, i.e., $\Sigma = \{1, \ldots, n^C\}$ for some constant C. We will later see that this assumption does not help in our case.

7.1.1 Maximal repetitions

For any string $w \in \Sigma^*$, an integer $1 \leq p < |w|$ is called a *period* of w if $w[i] = w[i+p]$ for all $1 \leq i \leq |w| - p$. A string whose smallest period is at most half its length is called a *repetition*. We are interested in occurrences of repetitions as a substring of a given string which are periodically maximal. Specifically, a triplet $r = (i, j, p)$ is called a *maximal repetition* of w, if and only if all the following hold:

1. p is the smallest period of $w[i..j]$ and $|w[i..j]| \geq 2p$ (repetition),
2. $i = 1$ or $w[i-1] \neq w[i-1+p]$ (left maximal), and
3. $j = |w|$ or $w[j+1] \neq w[j+1-p]$ (right maximal).

For any string w, we denote the set of maximal repetitions as $\mathit{MReps}(w)$. Although maximal repetitions are commonly referred to as "runs" in the literature, we use the term "maximal repetitions" so as not to confuse it with "run" in "run-length encoding".

For example, the string $w = \texttt{abaababaabaab}$ contains seven maximal repetitions, i.e., $\mathit{MReps}(w) = \{(3, 4, 1), (8, 9, 1), (11, 12, 1), (4, 8, 2), (1, 6, 3), (6, 13, 3), (1, 11, 5)\}$.

7.1.2 Run length encoding

Let $\mathcal{N}$ denote the set of positive integers. For any string $w \in \Sigma^*$, let $a_i \in \Sigma$ and $e_i \in \mathcal{N}$, for $1 \le i \le m$, be such that $w = a_1^{e_1} \cdots a_m^{e_m}$ and $a_i \ne a_{i+1}$ for all $1 \le i < m$. The *run-length encoding* $\mathit{RLE}(w)$ of string w is a string over the alphabet $\Sigma \times \mathcal{N}$, and is defined as $\mathit{RLE}(w) = (a_1, e_1) \cdots (a_m, e_m)$. For any $1 \le i \le m$, each letter $\mathit{RLE}(w)[i] = (a_i, e_i)$ and its corresponding substring $a_i^{e_i}$ in w is called a run-length factor, and e_i is called its exponent.

The set of starting (resp. ending) positions of run-length factors of w is denoted by S_w (resp. E_w), i.e., $S_w = \{1 + \sum_{k=1}^{i-1} e_k : 1 \le i \le m\}$ and $E_w = \{\sum_{k=1}^{i} e_k : 1 \le i \le m\}$. We will also write $S_w[i] = 1 + \sum_{k=1}^{i-1} e_k$ and $E_w[i] = \sum_{k=1}^{i} e_k$ for any $1 \le i \le m$.

7.1.3 Lyndon words

A string w is a *Lyndon word* [56] with respect to lexicographic order $\prec$, if and only if $w \prec w[i..|w|]$ for any $1 < i \le |w|$, i.e., w is lexicographically smaller than any of its proper suffixes with respect to $\prec$. It is easy to see that a Lyndon word w cannot have a non-empty border, since a border would be a proper suffix of w that is lexicographically smaller than w, since it is also a prefix of w. An equivalent definition for a Lyndon word, is a word which is lexicographically smaller than any of its proper cyclic rotations.

For example, if $\texttt{a} \prec \texttt{b}$, then, the string $\texttt{abaabb}$, $\texttt{baa}$, $\texttt{abab}$ are not Lyndon words with respect to $\prec$, while $\texttt{aabab}$ is. The following is also well known.

Lemma 26 (Proposition 1.3 [27]). *For any Lyndon words u and v, uv is a Lyndon word iff $u \prec v$.*

7.1.4 Longest common extension

For any string w of length n, the *longest common extension query* is, given two positions $1 \le i, j \le n$, to answer

$$\mathit{LCE}_w(i, j) = \max\{k \mid w[i..i + k - 1] = w[j..j + k - 1], i + k - 1, j + k - 1 \le n\}.$$

We also define the longest common extension in the reverse direction, i.e.,

$$LCE^R_w(i,j) = \max\{k \mid w[i-k+1..i] = w[j-k+1..j], i-k+1, j-k+1 \geq 1\}.$$

Note that if there is a way to compute $LCE_w(i,j)$ given w, there is also a way to compute $LCE^R_w(i,j)$ by considering the reversed string $w^R = w[n]\cdots w[1]$, since $LCE^R_w(i,j) = LCE_{w^R}(n-i+1, n-j+1)$.

7.2 The Maximum Number of Maximal Repetitions by RLE

The goal of this section is to prove the following Theorem.

Theorem 18. *For any string w, let m be the size of its run-length encoding, and k the number of run-length factors of w whose exponent is at least 2. Then, $|MReps(w)| \leq m + k - 1$.*

The proof basically follows the idea of [7] for normal strings, but it is extended to deal with RLE strings.

For any maximal repetition $r = (i,j,p)$ of string w and any lexicographic order $\prec$, there exists a substring of length p in $w[i..j]$ that is a Lyndon word with respect to $\prec$. This is because the set $\{w[i'..i'+p-1] \mid i+1 \leq i' \leq i+p\}$ contains all p cyclic rotations of $w[i+1..i+p]$ which are all distinct, since p is the smallest period of w, and a lexicographically smallest rotation will always exist. Any length p subinterval $[\ell, \ell+p-1]$ of a maximal repetition $r = (i,j,p)$ such that $w[\ell..\ell+p-1]$ is a Lyndon word with respect to $\prec$, is called an *L-root* of r with respect to $\prec$.

Theorem 18 is trivial when $|\Sigma| = 1$, so we can assume $|\Sigma| \geq 2$, and thus, we are able to consider two orderings denoted by $\prec_0$ and $\prec_1$, where $\prec_0 = \prec$ and for any $a, b \in \Sigma$, $a \prec_0 b \iff b \prec_1 a$. We also use $\prec_0$ and $\prec_1$ to denote the lexicographic orders on Σ^* induced by the respective total orders. As in [7], we choose, for each maximal repetition $r = (i,j,p)$, a specific lexicographic order denoted by $\prec_r \in \{\prec_0, \prec_1\}$ so that $w[j+1] \prec_r w[j+1-p]$. We note that either order can be chosen when $j = n$. The set B_r is defined as the beginning positions of L-roots of r with respect to this order, but excludes a position if it coincides with the beginning position of the maximal repetition, i.e., for any maximal repetition $r = (i,j,p)$,

$$B_r = \{\ell \mid [\ell..\ell+p-1] \text{ is an L-root of } r \text{ w.r.t. } \prec_r \text{, and } \ell \neq i\}.$$

Note that $|B_r| \geq 1$ since a maximal repetition always contains an L-root that does not start at its beginning. One of the crucial results of [7] was the following lemma, which implies that the number of maximal repetitions in a string w of length n is at most $n-1$ since $\cup_{r \in \mathit{MReps}(w)} B_r \subseteq [2..n]$ and thus $|\mathit{MReps}(w)| \leq \sum_{r \in \mathit{MReps}(w)} |B_r| \leq n-1$.

Lemma 27 (Lemma 8 of [7]). *For any distinct maximal repetitions r, r' of w, $B_r \cap B_{r'} = \emptyset$.*

The following lemma is an important new observation for L-roots of maximal repetitions with respect to their run-length encoding.

Lemma 28. *For any maximal repetition $r = (i, j, p)$ of string w with $p \geq 2$, it holds that $B_r \subset S_w$, i.e., a position in B_r must be the beginning of an RLE-factor.*

Proof. Suppose to the contrary, that there is some $\ell \in B_r$ that is not at the beginning of an RLE-factor, i.e., $\ell \notin S_w$, and let $[\ell..\ell+p-1]$ be the corresponding L-root of r. By the assumption, $w[\ell-1] = w[\ell]$. Furthermore, by the definition of B_r, we have that $i < \ell$ and by the periodicity of r, $w[\ell-1] = w[\ell+p-1]$. However, this implies that $w[\ell..\ell+p-1]$ has a border, contradicting that it is a Lyndon word. The lemma holds, since $1 \in S_w$ but $1 \notin B_r$.

Of course, a run-length factor can be a maximal repetition of period 1, and can be stated as follows.

Lemma 29. *For any string w, let $\mathit{RLE}(w) = (a_1, e_1) \cdots (a_m, e_m)$. For any $1 \leq i \leq m$, $(S_w[i], E_w[i], 1)$ is a maximal repetition of period 1 if and only if $e_i \geq 2$.*

We are now ready to prove Theorem 18.

Proof.[Proof of Theorem 18] Recall that k is the number of run-length factors of w whose exponent is at least 2. Due to Lemma 29, the number of maximal repetitions with period 1 is equal to k. Note that for any maximal repetition r of period 1, any position $i \in B_r$ satisfies $i \notin S_w$. Let $\mathit{MReps}_{p\geq 2}(w)$ be the set of maximal repetitions such that the period is at least 2. From $|B_r| \geq 1$ and Lemmas 27 and 28,

$$|\mathit{MReps}_{p\geq 2}(w)| \leq \sum_{r \in \mathit{MReps}_{p\geq 2}(w)} |B_r| \leq m - 1 < m = |S_w|$$

holds. Thus, the total number of maximal repetitions is at most $m + k - 1$.

If we consider the 2 cases w.r.t. m, we can get better bounds for each of 2 cases. Corollary 2 is the tight bound for smaller m. Corollary 3 is an improved bound for larger m.

Corollary 2. *For any string w, let m be the size of its run-length encoding. If $m \leq 3$, $|MReps(w)| \leq m$.*

If $m = 3$, it is easy to see that $|MReps_{p\geq 2}(w)| = 0$. Obviously, $|MReps(w)| = k$ also holds, where k is the number of run-length factors of w whose exponent is at least 2.

Corollary 3. *For any string w, let m be the size of its run-length encoding, and k the number of run-length factors of w whose exponent is at least 2. If $m \geq 4$, $|MReps(w)| \leq m + k - 3$.*

Proof. Since an L-root of $r \in MReps_{p\geq 2}(w)$ must contain at least two different characters, the beginning position $S_w[m]$ of the last run-length factor (a_m, e_m) cannot be in B_r for any $r \in MReps_{p\geq 2}(w)$. If $S_w[m-1] \in B_r$ for some $r \in MReps_{p\geq 2}(w)$, this implies that $[S_w[m-1], E_w[m]]$ is an L-root of r. Thus, $[S_w[m-2], E_w[m-1]]$ must also be an L-root with respect to the lexicographically reversed order, and $S_w[m-2] \notin B_r$. Since r ends at position $|w|$, we can choose either $S_w[m-1]$ or $S_w[m-2]$ as an element of B_r. This implies that either $S_w[m-1]$ or $S_w[m-2]$ is not in B_r for any $r \in MReps_{p\geq 2}(w)$. Thus $|MReps_{p\geq 2}(w)| \leq m - 3$ also holds. Since 1 and $S_w[m]$ are not contained in B_r, and since only one of $S_w[m-1]$ or $S_w[m-2]$ is contained in some B_r, we have that the maximum number of maximal repetitions in a string is at most $m + k - 3$.

7.3 Computing All Maximal Repetitions on RLE strings

In this section, we propose an algorithm to compute all maximal repetitions on RLE strings. Our algorithm follows the new algorithm for normal strings proposed in [7], but is modified to handle RLE strings. We first review the algorithm for non-RLE strings.

7.3.1 Overview of algorithm for non-RLE strings

The crucial observation made in [7] (which was also required for the proof of Lemma 27 in the previous section) is the following:

Lemma 30 (Lemma 7 of [7])**.** *For any maximal repetition $r = (i, j, p)$ of string w, let $[\ell, \ell + p - 1]$ be an L-root of r with respect to order $\prec_r$. Then, $w[\ell..\ell + p - 1]$ is the longest Lyndon word that is a prefix of $w[\ell..|w|]$.*

Based on this observation, the algorithm consists of two steps. Step 1: Compute all the longest Lyndon words with respect to $\prec_0$ and $\prec_1$ that start at each position of the string (the occurrences are candidates for L-roots). Step 2: For each such candidate $\lambda = w[i_\lambda..j_\lambda]$, compute $\ell_h = LCE_w(i_\lambda, j_\lambda{+}1)$ and $\ell_g = LCE^R_w(i_\lambda{-}1, j_\lambda)$ to see how long the period $p_\lambda = |w[i_\lambda..j_\lambda]| = j_\lambda - i_\lambda + 1$ continues to the left and to the right. We see that $[i_\lambda, j_\lambda]$ is indeed an L-root of the maximal repetition $r = (i_\lambda - \ell_g, j_\lambda + \ell_h, p_\lambda)$ if and only if $\ell_g + \ell_h \geq p_\lambda$.

Noticing that a Lyndon word can be created from any string by appending a unique smallest letter to the front of the string, we can use the *Lyndon tree* of a Lyndon word for Step 1. Given a Lyndon word w of length $n > 1$, (u, v) is the *standard factorization* [17, 55] of w, if $w = uv$ and v is the longest proper suffix of w that is a Lyndon word, or equivalently, the lexicographically smallest proper suffix of w. It is well known that for the standard factorization (u, v) of any Lyndon word w, the factors u and v are also Lyndon words (e.g.[8]). The *Lyndon tree* of w is the full binary tree defined by recursive standard factorization of w; w is the root of the Lyndon tree of w, its left child is the root of the Lyndon tree of u, and its right child is the root of the Lyndon tree of v. The longest Lyndon word that starts at each position can be obtained from the Lyndon tree, due to the following lemma.

Lemma 31 (Lemma 22 of [7]). *Let w be a Lyndon word with respect to $\prec$. $w[i..j]$ corresponds to a right node (or possibly the root) of the Lyndon tree with respect to $\prec$ if and only if $w[i..j]$ is the longest Lyndon word with respect to $\prec$ that starts from i.*

The Lyndon tree of a normal string can be computed in $O(n\mathcal{A}(n))$ time over general ordered alphabet because of the following lemmas.

Lemma 32 (Observation 4 of [22]). *The Lyndon tree of a string of length n can be constructed by using $O(n)$ non-crossing LCE queries.*

Lemma 33 (Theorem 12 of [22]). *In a string of length n, a sequence of q non-crossing LCE queries can be answered in time $O(q + n\mathcal{A}(n))$, where $\mathcal{A}$ denotes the inverse Ackermann function.*

Here, a set of LCE queries is *non-crossing* if there are no two queries (i, j) and (i', j'), such that $i < i' < j < j'$ or $i' < i < j' < j$. After computing the Lyndon tree, $O(n)$ non-crossing LCE queries are computed again for each right node in Step 2 as described above. Thus the total time complexity for computing all maximal repetitions in non-RLE string is $O(n\mathcal{A}(n))$ time over general ordered alphabet.

We note that the LCE queries and thus all maximal repetitions can be computed in total $O(n)$ time for integer alphabets (using e.g. [30]).

7.3.2 Extending Lyndon structures for RLE

We now consider computing maximal repetitions on RLE strings. By Theorem 18, the number of maximal repetitions in an RLE string is $O(m)$, and from Lemmas 28 and 30, we can limit the candidate L-roots of maximal repetitions with period at least 2, to the longest Lyndon words that start at beginning positions of a run-length factor. We propose the *RLE-Lyndon tree* of a string which can be represented in $O(m)$ space and contains this information. In the RLE-Lyndon tree, we treat each run-length factor like a character. The idea of the extension comes from the following lemma.

Lemma 34. *For any* $1 \leq i < j \leq |w|$, *if* $w[i..j]$ *is the longest Lyndon word with respect to* $\prec$ *that is a prefix of* $w[i..|w|]$, *then* $j \in E_w$, *i.e.,* j *is an end of an RLE-factor.*

Proof. Suppose to the contrary, that there is some $j \notin E_w$ such that $w[i..j]$ is the longest Lyndon word with respect to $\prec$ that is a prefix of $w[i..|w|]$. Let $RLE(w)[k]$ be the run-length factor such that $S_w[k] \leq j < E_w[k]$. Since $w[i..j]$ is a Lyndon word of length at least 2 and $w[j] = a_k = w[j+1]$, $w[i..j] \prec w[j] = w[j+1]$ holds. By Lemma 26, $w[i..j+1]$ is also a Lyndon word. This contradicts that $w[i..j]$ is the longest Lyndon word with respect to $\prec$ that is a prefix of $w[i..|w|]$.

From Lemmas 28 and 34, we have that for any maximal repetition r, each L-root of r that has a starting position in B_r, starts at the beginning position of some run-length factor and ends at the ending position of some run-length factor. We note that RLE-Lyndon substring and RLE-Lyndon factorization which will be defined in this section were introduced in [37] in a different context.

Definition 12 (RLE-Lyndon substring)**.** *A string* x *is an* RLE-Lyndon substring *of* w *if* x *is a Lyndon word that is a concatenation of consecutive run-length factors of* w, *or* x *is a run-length factor.*

Definition 13 (RLE-standard factorization)**.** *A pair of strings* (u, v) *is an* RLE-standard factorization *of* w *if* $w = uv$ *and* v *is the longest proper suffix of* w *that is an RLE-Lyndon substring.*

Definition 14 (RLE-Lyndon tree)**.** *The* RLE-Lyndon tree *of a Lyndon word w, denoted $LyndonTre^2(w)$, is an ordered full binary tree defined recursively as follows:*

- *if $|RLE(w)| = 1$, then $LyndonTre^2(w)$ consists of a single node labeled by (a_1, e_1);*
- *if $|RLE(w)| \geq 2$, then the root of $LyndonTre^2(w)$, labeled by $RLE(w)$, has left child $LyndonTre^2(u)$ and right child $LyndonTre^2(v)$, where (u, v) is the RLE-standard factorization of w.*

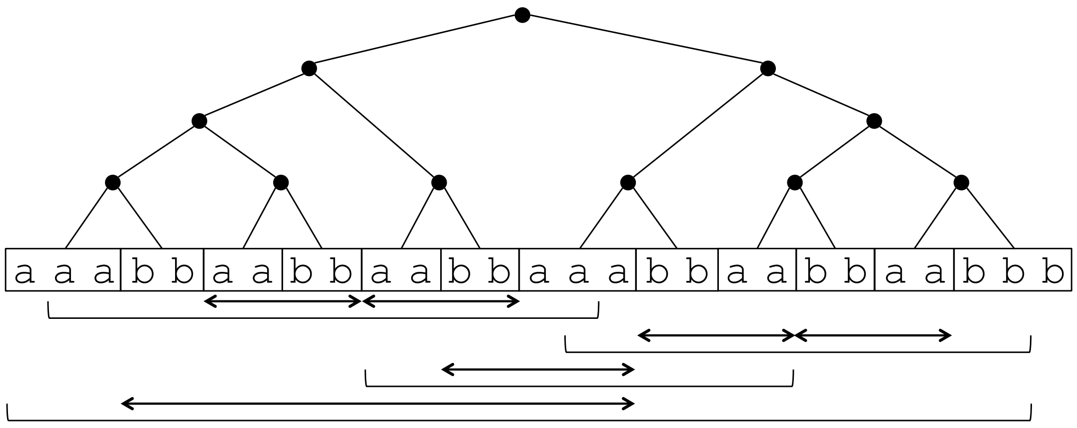


Figure 7.1: The RLE-Lyndon tree for the Lyndon word $\mathtt{a}^3\mathtt{b}^2\mathtt{a}^2\mathtt{b}^2\mathtt{a}^2\mathtt{b}^2\mathtt{a}^3\mathtt{b}^2\mathtt{a}^2\mathtt{b}^2\mathtt{a}^2\mathtt{b}^3$ with respect to order $\mathtt{a} \prec \mathtt{b}$. The double-headed arrow shows the L-roots that start at a position in B_r,for all 4 maximal repetitions with period at least 2.

Figure 7.1 shows the RLE-Lyndon tree of a string $\mathtt{a}^3\mathtt{b}^2\mathtt{a}^2\mathtt{b}^2\mathtt{a}^2\mathtt{b}^2\mathtt{a}^3\mathtt{b}^2\mathtt{a}^2\mathtt{b}^2\mathtt{a}^2\mathtt{b}^3$. Though the above structures are simply extended to RLE, it is interesting to note that these structures satisfy similar properties of the original structures. The most important property of the RLE-Lyndon tree in this chapter is stated in Lemma 35, which is an analogous to Lemma 31. The lemma can be shown by similar arguments as in [7].

Lemma 35. *Let w be a Lyndon word with respect to $\prec$. For any $i \in S_w$, $w[i..j]$ corresponds to a right node (or possibly the root) of $LyndonTre^2(w)$ with respect to $\prec$ if and only if $w[i..j]$ is the longest Lyndon word with respect to $\prec$ that starts from i.*

From the above lemma, we can detect all maximal repetitions in $MReps_{p\geq 2}(w)$ if we have $LyndonTre^2(w)$ (maximal repetitions with period 1 correspond to run-length factor or leaves of $LyndonTre^2(w)$). In the example of Figure 7.1, for each maximal repetition r, the L-roots that start at a position in B_r are drawn by double-headed arrows. For example, the 2 L-roots

$[S_w[3]..E_w[4]]$ and $[S_w[5]..E_w[6]]$ (corresponding to a Lyndon word `aabb`) with respect to the same order $\prec$ as the Lyndon tree is represented by an internal node which is a right child. Also, it can be observed that each L-root begins at the starting position of a run-length factor and ends at the ending position of a run-length factor of w.

In Section 7.3.3, we show an algorithm to compute $\mathit{LyndonTre}^2(w)$. For convenience, we present the notion of *RLE-Lyndon factorizations* and show some properties of RLE-Lyndon factorizations.

Definition 15 (RLE-Lyndon factorization). *A sequence $w_1, \ldots, w_s$ is the RLE-Lyndon factorization of w if each w_i is an RLE-Lyndon substring, $w_1 \succeq \ldots \succeq w_s$, and $w = w_1 \cdots w_s$.*

The difference between the original Lyndon factorization [17] and the RLE-Lyndon factorization arises for Lyndon factors which are a single letter in the original Lyndon factorization. For a string $w =$ `bbbabbaabbaa`, the original Lyndon factorization of w is $\mathtt{b} \succeq \mathtt{b} \succeq \mathtt{b} \succeq \mathtt{abb} \succeq \mathtt{aabb} \succeq \mathtt{a} \succeq \mathtt{a}$, the RLE-Lyndon factorization of w is $\mathtt{b}^3 \succeq \mathtt{abb} \succeq \mathtt{aabb} \succeq \mathtt{a}^2$. Thus similar argument about the longest Lyndon word on Lyndon factorizations holds, as below.

Lemma 36. *Let $w_1, \ldots, w_s$ be the RLE-Lyndon factorization of w. Then, w_1 is either $\mathit{RLE}(w)[1]$ or the longest Lyndon word that is a prefix of w.*

This implies that w_i is either $\mathit{RLE}(w_i \cdots w_s)[1]$ or the longest Lyndon word that is a prefix of $w_i \cdots w_s$.

7.3.3 Algorithms

Finally, we show how to compute $\mathit{LyndonTre}^2(w)$ in $O(m\mathcal{A}(m))$ time and $O(m)$ space. After we compute $\mathit{LyndonTre}^2(w)$, we can compute all maximal repetitions by using non-crossing LCE queries. Note that the $O(n)$ time and space solution for non-RLE strings over the integer alphabet cannot be applied to $\mathit{RLE}(w)$ to achieve an $O(m)$ time and space solution, since the alphabet for $\mathit{RLE}(w)$ cannot be assumed to be an integer alphabet in terms of its length m (m could be much smaller than n, while an exponent of a run-length factor could be as large as n). However, the solution to non-crossing LCE queries for non-RLE strings over a general ordered alphabet can be easily extended to LCE queries on an RLE string, since the algorithm is based only on character comparisons.

Corollary 4. *For any RLE string $\mathit{RLE}(w)$ of size m, a sequence of q non-crossing LCE queries on $\mathit{RLE}(w)$ can be answered in time $O(q + m\mathcal{A}(m))$.*

We use the above corollary in order to decide the lexicographic order between RLE substrings in the construction of $\mathit{LyndonTre}^2(w)$, and to compute maximal repetitions.

Lemma 37. *$\mathit{LyndonTre}^2(w)$ can be computed in $O(m\mathcal{A}(m))$ time and $O(m)$ space.*

Proof. Firstly, we show our algorithm. The algorithm constructs $\mathit{LyndonTre}^2(w)$ in bottom-up and from right to left. The main idea is that the right factor of RLE-standard factorization is the longest proper suffix which is an RLE-Lyndon substring. We will find such a suffix by concatenating two RLE-Lyndon substrings based on Lemma 26. Since each leaf corresponds to a single run-length factor (i.e., RLE-Lyndon substring), we know that the tree has m leaves. A stack is maintained so that at the beginning of k-th step, the stack contains the sequence of subtrees of $\mathit{LyndonTre}^2(w)$ such that the corresponding sequence of RLE-Lyndon substrings is the RLE-Lyndon factorization of the suffix $w[S_w[m-k+2]..|w|]$. In the k-th step, the algorithm pushes the leaf corresponding to $\mathit{RLE}(w)[m-k+1]$ on the stack. Let (f_b, f_e) (resp. (s_b, s_e)) be pair of positions in $\mathit{RLE}(w)$ such that the top (resp. second) subtree in the stack corresponds to the RLE-Lyndon substring $w[S_w[f_b]..E_w[f_e]]$ (resp. $w[S_w[s_b]..E_w[s_e]]$). Note that $E_w[f_e]+1 = S_w[s_b]$ always holds. After pushing the new leaf, the algorithm does the following;

- If $w[S_w[f_b]..E_w[f_e]] \prec w[S_w[s_b]..E_w[s_e]]$, pop the two elements and push the subtree which is the concatenation of the two popped subtrees, and repeat the process.
- Otherwise, go to the next step.

We now prove that the above invariant condition of the stack holds before $k+1$-th step. We denote the RLE-Lyndon factorization of the suffix $w[S_w[m-k+2]..|w|]$ by $W_1, \ldots, W_j$. Because of the above operations, a factorization of the suffix $w[S_w[m-k+1]..|w|]$ can be represented by $W', W_i, \ldots, W_j$ for some $1 \le i \le j$ where $W' = \mathit{RLE}(w)[m-k+1]W_1 \cdots W_{i-1}$ (for convenience, $W_0 = \epsilon$). By the assumption, $W_i, \ldots, W_j$ is the RLE-Lyndon factorization of the suffix $W_i \cdots W_j$. By the algorithm and Lemma 26, W' is an RLE-Lyndon substring and $W' \succeq W_i$ holds. Thus $W', W_i, \ldots, W_j$ is the RLE-Lyndon factorization of the suffix $w[S_w[m-k+1]..|w|]$ since $W' \succeq W_i \succeq \ldots \succeq W_j$ holds. Since w is a Lyndon word, when all leaves are pushed on the stack and the number of elements in the stack is one, the algorithm stops and the RLE-Lyndon tree is completely constructed.

We can determine the lexicographic order by using LCE queries. More precisely, for each lexicographic comparison described above, we compute $\mathit{LCE}_{\mathit{RLE}(w)}(f_b, s_b) = k$. Then, $w[S_w[f_b]..E_w[f_e]] \prec w[S_w[s_b]..E_w[s_e]]$ if and only if $s_b + k - 1 < s_e$ and, either

1. $a_{f_b+k} \prec a_{s_b+k}$ or, $a_{f_b+k} = a_{s_b+k}$ and
2. $e_{f_b+k} < e_{s_b+k}$ and $a_{f_b+k+1} \prec a_{s_b+k}$ or
3. $e_{f_b+k} > e_{s_b+k}$ and $a_{s_b+k+1} \prec a_{f_b+k}$.

Thus, in the algorithm, we call $O(m)$ non-crossing LCE queries such that each query positions is the beginning position of some run-length factor and we can compute $\mathit{LyndonTre}^2(w)$ in $O(m\mathcal{A}(m))$ time.

To compute all maximal repetitions, we need to compute another $O(m)$ sets of LCE queries on w (or w^R) for each candidate L-root. The query positions are starting positions of run-length factors in w (or w^R). It is easy to see that this can also be achieved in $O(m\mathcal{A}(m))$ time by Corollary 4 since if $k = \mathit{LCE}_{\mathit{RLE}(w)}(i, j)$, then $\mathit{LCE}_w(S_w[i], S_w[j]) = E_w[i + k - 1] - S_w[i] + 1 + e$, where $e = \min\{e_{i+k}, e_{j+k}\}$ if $a_{i+k} = a_{j+k}$ and 0 otherwise. It is also clear that the algorithm requires $O(m)$ space.

Therefore, the following theorem holds.

Theorem 19. *Given a run-length encoding of a string w, all maximal repetitions in w can be computed in $O(m\mathcal{A}(m))$ time and $O(m)$ space.*

Corollary 5. *For any string w, let $\mathit{RLE}(w) = (a_1, e_1) \cdots (a_m, e_m)$. If for all $1 \le i \le m$, $a_i \in \{1, \ldots, m^{C_1}\}$, and $e_i = O(m^{C_2})$ for some constants C_1 and C_2, then all maximal repetitions in w can be computed in $O(m)$ time and $O(m)$ space.*

Proof. Under the assumption, any set of $O(m)$ LCE queries on $\mathit{RLE}(w)$ can be answered in $O(m)$ total time using the methods for integer alphabets (i.e., $\Sigma = \{1, \ldots, m^C\}$ for some constant C), since $a_i^{e_i} = O(m^{C_1+C_2})$.

Chapter 8

Conclusion

In this thesis, we studied compact string indexing and their applications to efficient pattern discovery by using the combinatorial properties of strings.

In Chapter 3, we proposed the first $O(n)$-time algorithm to construct edge-sorted DAWGs for integer alphabets. A simple modification also leads us to the first $O(n)$-time algorithm to construct affix trees for integer alphabets. We remark that the previously best known DAWG (resp. affix-tree) construction algorithm of Blumer et al. (resp. Maaß) requires $O(n \log n)$ time for integer alphabets.

In Chapter 4, we proposed a new space saving data structure called the truncated DAWGs. We show that the k-truncated DAWG of y, denoted by $k\text{-}TDAWG(y)$, is a subgraph of $DAWG(y)$, and can be stored in $O(\min\{n, k\gamma\})$ space where n is the length of y, σ is the alphabet size, and γ is the size of one of the smallest k-attractors of y. We also presented an $O(n \log \sigma)$ time and $O(\min\{n, k\gamma\})$ space algorithm for constructing $k\text{-}TDAWG(y)$. As an application of $k\text{-}TDAWG(y)$, we presented an $O(\min\{n, k\gamma\} + |MAW_k(y)|)$ time algorithm to compute the set $MAW_k(y)$ of all minimal absent words of y whose size is smaller than or equal to k by using $k\text{-}TDAWG(y)$.

In Chapter 5, we addressed the left-right maximal generic words problem and developed an $O(n \log m)$ size data structure, which answers queries in $O(|p| + o \log\log m)$ time, where o is the size of outputs. Our method is better than the previous work by Nishimoto et al. both in the space requirement and in the query time.

In Chapter 6, we presented an online algorithm which enumerates all length-constrained gapped palindromes occurring in a string w of length n in $O(n(\frac{g_{\min} - g_{\max}}{A} + \log \sigma))$ time, for given parameters $2 \leq g_{\min} \leq g_{\max}$ and $A \geq 1$. We also showed that if A is a constant, then there exists a string which contains $\Omega((g_{\min} - g_{\max})n)$ maximal LCGPs. This implies that for a constant-size alphabet the running time of our algorithm is optimal in the worst case.

In Chapter 7, we considered maximal repetitions on run length encoded strings. Firstly, we presented a new upper bound of maximal repetition in string w $2m-1$ where m is the size of run length encoded string of w. Secondly, we presented an algorithm to enumerate all maximal repetitions in $O(m\mathcal{A}(m))$ time, where $\mathcal{A}$ denotes the inverse Ackermann function. Our algorithm is faster than or equal to Chrochemore et al.'s algorithm (for an uncompressed string) even if including the time for computing the run length encoding of uncompressed input string.

Bibliography

[1] Pizza&Chili Corpus. `http://pizzachili.dcc.uchile.cl/texts/nlang/`.

[2] *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013*. IEEE, 2013.

[3] A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. *Theor. Comput. Sci.*, 141(1&2):163–173, 1995.

[4] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *SPIRE 2012*, pages 61–72, 2012.

[5] A. Bagnall, C. A. Ratanamahatana, E. Keogh, S. Lonardi, and G. Janacek. A bit level representation for time series data mining with shape based similarity. *Data Mining and Knowledge Discovery*, 13(1):11–40, 2006.

[6] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. A new characterization of maximal repetitions by Lyndon trees. In P. Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 562–571. SIAM, 2015.

[7] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017.

[8] F. Bassino, J. Clément, and C. Nicaud. The standard factorization of Lyndon words: an average point of view. *Discrete Mathematics*, 290(1):1–25, 2005.

[9] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In G. H. Gonnet, D. Panario, and A. Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[10] O. Berkman and U. Vishkin. Finding Level-Ancestors in Trees. *J. Comput. Syst. Sci.*, 48(2):214–230, 1994.

[11] S. Biswas, M. Patil, R. Shah, and S. V. Thankachan. Succinct indexes for reporting discriminating and generic words. In E. S. de Moura and M. Crochemore, editors, *String Processing and Information Retrieval - 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20-22, 2014. Proceedings*, volume 8799 of *Lecture Notes in Computer Science*, pages 89–100. Springer, 2014.

[12] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoreoretical Computer Sceince*, 40:31–55, 1985.

[13] A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[14] S. Chairungsee and M. Crochemore. Using minimal absent words to build phylogeny. *Theor. Comput. Sci.*, 450:109–116, 2012.

[15] K. Chen and K. Chao. A fully compressed algorithm for computing the edit distance of run-length encoded strings. *Algorithmica*, 65(2):354–370, 2013.

[16] K. Chen, P. Hsu, and K. Chao. Efficient retrieval of approximate palindromes in a run-length encoded string. *Theor. Comput. Sci.*, 432:28–37, 2012.

[17] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus. iv. the quotient groups of the lower central series. *Annals of Mathematics*, 68(1):81–95, 1958.

[18] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005.

[19] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.

[20] M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis, and Y. Ramusat. Minimal absent words in a sliding window and applications to on-line pattern matching. In *Fundamentals of Computation Theory - 21st International Symposium, FCT 2017, Bordeaux, France, September 11-13, 2017, Proceedings*, pages 164–176, 2017.

[21] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008.

[22] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In *Proc. SPIRE 2016*, pages 22–34, 2016.

[23] M. Crochemore, F. Mignosi, and A. Restivo. Automata and forbidden words. *Information Processing Letters*, 67(3):111–117, 1998.

[24] M. Crochemore and D. Perrin. Two-way string matching. *J. ACM*, 38(3):651–675, 1991.

[25] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Publishing Co. Pte. Ltd., 2002.

[26] N. G. De Bruijn. A combinatorial problem. In *Proc. Koninklijke Nederlandse Academie van Wetenschappen*, volume 49, pages 758–764, 1946.

[27] J.-P. Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983.

[28] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[29] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16:109–114, 1965.

[30] J. Fischer and V. Heun. Theoretical and practical improvements on the rmq-problem, with applications to LCA and LCE. In M. Lewenstein and G. Valiente, editors, *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[31] Y. Fujishige, T. Takagi, and D. Hendrian. Truncated DAWGs and Their Application to Minimal Absent Word Problem. In *25th International Symposium on String Processing and Information Retrieval*, volume 4209, pages 139–152. Springer International Publishing, 2018.

[32] Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda. Computing dawgs and minimal absent words in linear time for integer alphabets. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, pages 38:1–38:14, 2016.

[33] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Efficiently finding all maximal alpha-gapped repeats. In N. Ollinger and H. Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[34] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory Comput. Syst.*, 62(1):162–191, 2018.

[35] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen. Faster longest common extension queries in strings over general alphabets. In *Proc. CPM 2016*, pages 5:1–5:13, 2016.

[36] P. Gawrychowski, G. Kucherov, Y. Nekrich, and T. A. Starikovskaya. Minimal discriminating words problem revisited. In O. Kurland, M. Lewenstein, and E. Porat, editors, *String Processing and Information Retrieval - 20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7-9, 2013, Proceedings*, volume 8214 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 2013.

[37] S. S. Ghuman, E. Giaquinta, and J. Tarhio. Alternative algorithms for lyndon factorization. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*, pages 169–178. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2014.

[38] A. Golynski, J. I. Munro, and S. S. 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. ACM Press, 2006.

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

[40] S. Inenaga. Bidirectional construction of suffix trees. *Nord. J. Comput.*, 10(1):52, 2003.

[41] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005.

[42] D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In I. Diakonikolas, D. Kempe, and M. Henzinger, editors, *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 827–840. ACM, 2018.

[43] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[44] R. Kolpakov, G. Bana, and G. Kucherov. mreps: Efficient and flexible detection of tandem repeats in DNA. *Nucleic acids research*, 31(13):3672–3678, July 2003.

[45] R. Kolpakov and G. Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009.

[46] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999.

[47] D. Kosolobov. Lempel-ziv factorization may be harder than computing all runs. In E. W. Mayr and N. Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015, March 4-7, 2015, Garching, Germany*, volume 30 of *LIPIcs*, pages 582–593. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[48] D. Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016.

[49] D. Kosolobov, M. Rubinchik, and A. M. Shur. Finding distinct subpalindromes online. In *PSC 2013*, pages 63–69, 2013.

[50] K. Kuboi, Y. Fujishige, S. Inenaga, H. Bannai, and M. Takeda. Faster STR-IC-LCS computation via RLE. In *Combinatorial Pattern Matching, 28th Annual Symposium, CPM 2017, Warsaw, Poland, July 4-6, 2017, Proceedings*, 2017. in press.

[51] G. Kucherov, Y. Nekrich, and T. A. Starikovskaya. Computing discriminating and generic words. In L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, editors, *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 307–317. Springer, 2012.

[52] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144, 2007.

[53] Y. Lin, C. B. Ward, B. Jain, and S. Skiena. Constructing orthogonal de bruijn sequences. In F. Dehne, J. Iacono, and J. Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 595–606. Springer, 2011.

[54] J. J. Liu, Y. Wang, and Y. Chiu. Constrained longest common subsequences with run-length-encoded strings. *Comput. J.*, 58(5):1074–1084, 2015.

[55] M. Lothaire. *Combinatorics on Words*. Addison-Wesley, 1983.

[56] R. C. Lyndon. On Burnside's problem. *Transactions of the American Mathematical Society*, 77(2):202–202, Feb. 1954.

[57] M. G. Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.

[58] M. G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[59] M. G. Main and R. J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.

[60] G. K. Manacher. A new linear-time on-line algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975.

[61] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[62] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[63] F. Mignosi, A. Restivo, and M. Sciortino. Words and forbidden factors. *Theor. Comput. Sci.*, 273(1-2):99–117, 2002.

[64] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In D. Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA.*, pages 657–666. ACM/SIAM, 2002.

[65] J. C. Na, A. Apostolico, C. S. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 1-3(304):87–101, 2003.

[66] K. Narisawa, S. Inenaga, H. Bannai, and M. Takeda. Efficient computation of substring equivalence classes with suffix arrays. In *Proc. CPM 2007*, pages 340–351, 2007.

[67] T. Nishimoto, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Computing left-right maximal generic words. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24-26, 2015*, pages 5–16. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2015.

[68] N. Prezza. String attractors. *CoRR*, abs/1709.05314, 2017.

[69] J. Stoye. Affix trees. Technical Report Report 2000-04, Universität Bielefeld, 2000.

[70] S. Sugimoto, S. Inenaga, H. Bannai, and M. Takeda. Finding absent words from grammar compressed strings. In the Festschrift for Bořivoj Melichar, 2012.

[71] Y. Tanimura, Y. Fujishige, T. I, S. Inenaga, H. Bannai, and M. Takeda. A faster algorithm for computing maximal α-gapped repeats in a string. In *SPIRE 2015*, pages 124–136, 2015.

[72] Y. Tanimura, T. Nishimoto, H. Bannai, S. Inenaga, and M. Takeda. Small-space LCE data structure with constant-time queries. In *42nd International Symposium on Mathematical*

Foundations of Computer Science, MFCS 2017, August 21-25, 2017 - Aalborg, Denmark, pages 10:1–10:15, 2017.

[73] A. Thue. Über unendliche zeichenreihen. *Christiana Videnskabs Selskabs Skrifter, I. Math. naturv. Klasse, 7*, 1906.

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

[75] P. 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, 1973.

[76] J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda. Faster compact on-line Lempel-Ziv factorization. In *STACS 2014*, pages 675–686, 2014.

[77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.