Space-Efficient Algorithms for Computing Unique Substrings

三重野, 琢也

出版情報：九州大学, 2021, 博士（情報科学）, 課程博士
バージョン：
権利関係：

Space-Efficient Algorithms for Computing Unique Substrings

Takuya Mieno

July, 2021

Abstract

With the rapid development of network and sensor technologies in recent years, a large amount of digital data is being generated every day. The importance of creating new values by analyzing such large-scale data has been pointed out in many fields. However, when the size of the data is huge, efficient processing becomes difficult because of the pressure on memory and communication bandwidth. Therefore, it is important to develop *space-efficient* data processing methods for large-scale data. Any computer-readable digital data can be represented as a sequence of symbols, i.e., a string. In this thesis, we develop space-efficient data structures and algorithms for string data processing and clarify combinatorial properties on strings for the developments.

A substring of a string T that occurs exactly once in T is said to be *unique* in T. In this thesis, we focus on the problems of computing unique substrings, and aim to (A) clarify combinatorial properties on unique substrings, (B) develop space-efficient data structures for computing unique substrings, and (C) develop algorithms for computing unique substrings in semi-dynamic strings.

(A) We study the number of *shortest unique substrings* (*SUSs*) in a string and the relationship between run-length encoded strings and *minimal unique substrings* (*MUSs*) in a string. For a string T, a unique substring u of T is called a MUS of T if any shorter substring of u is not unique in T. Also, for an interval $[s, t]$ in T, a unique substring v of T is called a SUS of T for interval $[s, t]$ if v contains interval $[s, t]$ and any shorter substring of v containing $[s, t]$ is not unique in T. We show the tight bounds on the maximum of the total number of substrings which can be a SUS for some text position in a string. This is the first non-trivial result for combinatorial properties on SUSs. We next focus on the well-known string compression method run-length encoding (RLE), and study relations between run-length encoded strings and unique substrings. We show that the number m of MUSs of a string is less than twice the size r of the string compressed with RLE. Furthermore, we show the upper bound is tight, that is, there exists a family of strings that satisfies $m = 2r - 1$.

(B) We propose space-efficient data structures for computing SUSs based on RLE or tech-

niques of succinct data structures. Given a query interval $[s, t]$ in a string T of length n, the interval SUS problem is to output all the SUSs of T for $[s, t]$. When $s = t$, a query $[s, t]$ refers to a single position in T, and the problem is specifically called the point SUS problem. For the interval SUS problem, we propose a new data structure of size $O(r)$ where r is the RLE size of the input string T. This is based on the aforementioned result of the number of MUSs in RLE strings. All known methods for solving SUS problems require $O(n)$ words of space. We emphasize that the size $O(r)$ of our data structure is not worse than $O(n)$ since $r \leq n$ holds for any string, and thus, it can be sub-linear when the input string is well-compressible by RLE. We also give an alternative data structure of size $4n + o(n)$ bits for the problem utilizing succinct data structures. Furthermore, we give a smaller data structure of size $2.6n + o(n)$ bits for the point SUS problem. Both of these $O(n)$ bits data structures can be constructed in $O(n)$ time, and the working space is also small.

(C) We address the problem of computing MUSs and minimal unique palindromic substrings (MUPSs) in semi-dynamic strings. For a string T, a unique substring u of T is called a MUPS of T if u is a palindrome and any shorter palindromic substring of u is not unique in T. Our problems aim to maintain MUSs and MUPSs in a *semi-dynamic* string, where we can append a character to the right-end of the string or delete the left-most character from the string. We propose algorithms for maintaining MUSs and MUPSs in a semi-dynamic string running in amortized $O(\log \sigma)$ time for each append / delete operation, using $O(n)$ space where σ is the size of the alphabet. For the static setting, linear-time algorithms for computing MUSs and MUPSs of a string over an integer alphabet are known. Our algorithms are the first results to compute them efficiently when both sides of a given string are dynamically changing. As a bonus, we also propose a sliding-window-algorithm for maintaining a data structure called the palindromic tree (a.k.a. eertree) of a string that stores all distinct palindromes in the string.

Acknowledgments

I would like to express my gratitude to everyone who supported my research life at Kyushu University.

First of all, I would like to show my greatest appreciation to Professor Masayuki Takeda, my supervisor, and my thesis committee member. He taught me about the attitude towards research and the fun of research activities. I would also like to express my appreciation to Professor Eiji Takimoto, my thesis committee chair, and Associate Professor Daisuke Ikeda, my thesis committee member.

I also thank all of the staff in Department of Informatics, Kyushu University for their generous support. I would particularly like to thank Professor Hideo Bannai, Associate Professor Shunsuke Inenaga, Assistant Professor Yuto Nakashima, and Assistant Professor Dominik Köppl. They taught me how to do research and gave me a great deal of knowledge and many valuable ideas. I would also like to thank the work of the past and present members of our laboratory.

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 MFCS'16, the Proc. of CPM'17, the Proc. of SPIRE'19, and the Proc. of SOFSEM'20. Also, the journal version of SPIRE'19 was published in Theoretical Computer Science by Elsevier. I am thankful for all editors, committees, anonymous referees, and publishers.

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

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 Shortest Unique Substring Problems 1

1.2 Our Contributions 3

1.3 Organization 4

2 Preliminaries **6**

2.1 Notations 6

2.2 Algorithmic Tools 8

3 Tight Bounds on the Maximum Number of Shortest Unique Substrings **12**

3.1 Preliminaries 13

3.2 Bounds on the Number of Point SUSs 13

3.3 Bounds on the Number of Interval SUSs 21

3.4 Conclusions and Open Questions 21

4 Shortest Unique Substring Queries on Run-Length Encoded Strings **23**

4.1 Preliminaries 24

4.2 Computing MUSs from RLE Strings 27

4.3 Solution to the SUS Problem 30

4.4 Conclusions and Open Question 32

5 Space-Efficient Algorithms for Computing Minimal/Shortest Unique Substrings **33**

5.1 Computing MUSs in Compact Space 33

5.2 Compact Data Structure for the Interval SUS Problem 35

5.3 Compact Data Structure for the Point SUS Problem 37
5.4 Auxiliary Data Structure 48
5.5 Conclusions 49

6 Computing Minimal Unique Substrings for a Semi-Dynamic String 51
6.1 Preliminaries 52
6.2 Combinatorial Results on MUSs for a Sliding Window 53
6.3 Algorithm for computing MUSs for a Sliding Window 59
6.4 Conclusions and Future Work 69

7 Computing Minimal Unique Palindromic Substrings for a Semi-Dynamic String via Palindromic Tree 71
7.1 Preliminaries 73
7.2 Combinatorial Properties on Palindromes for a Sliding Window 74
7.3 Eertree for a Sliding Window 77
7.4 Applications of Eertrees for a Sliding Window 83
7.5 Conclusions and Future Work 85

8 Conclusions 87

Chapter 1

Introduction

With the rapid development of network and sensor technologies in recent years, a large amount of digital data is being generated every day. The importance of creating new values by analyzing such large-scale data has been pointed out in many fields. However, when the size of the data is huge, efficient processing becomes difficult because of the pressure on memory and communication bandwidth. Therefore, it is important to develop *space-efficient* data processing methods for large-scale data. Any computer-readable digital data can be represented as a sequence of symbols, i.e., a string. In this thesis, we develop space-efficient data structures and algorithms for string data processing and clarify combinatorial properties on strings for the developments.

A substring of a string T that occurs exactly once in T is said to be *unique* in T. In this thesis, we focus on the problems of computing unique substrings, and aim to (A) clarify combinatorial properties on unique substrings, (B) develop space-efficient data structures for computing unique substrings, and (C) develop algorithms for computing unique substrings in semi-dynamic strings.

1.1 Shortest Unique Substring Problems

First of all, we show the definition of our main problem; *shortest unique substring problem*, as well as related work on the problem. A substring $u = T[i..j]$ of a string T is called a *shortest unique substring (SUS)* for an interval $[s, t]$ if (a) u occurs exactly once in T, (b) u contains the interval $[s, t]$ (i.e., $i \leq s \leq t \leq j$), and (c) every substring v of T with $|v| < |u|$ containing $[s, t]$ occurs at least twice in T. Given a query interval $[s, t] \subset [0, n-1]$, the *interval SUS problem* is to output all the SUSs for $[s, t]$. When a query interval consists of a single position (i.e., $s = t$), the SUS problem becomes the so-called *point* SUS problem.

Point SUS Problems. The point SUS problem was introduced by Pei et al. [49]. This problem is motivated by applications in bioinformatics like genome comparisons [25] or PCR primer design [49]. Pei et al. tackled this problem with an $O(n)$ words data structure that can return one SUS for a given query position in constant time. They can compute this data structure in $O(n^2)$ time with $O(n)$ space. Based on that result, Tsuruta et al. [55] provided an $O(n)$ words data structure answering the same query (returning one SUS) in constant time. Their data structure can be constructed in $O(n)$ time. İleri et al. [28] independently showed another data structure with the same time complexities. For a general point SUS problem, Tsuruta et al. [55] can also resort to their proposed data structure returning all SUSs for a query position in optimal $O(k)$ time, where k is the number of returned SUSs.

The aforementioned data structures all take $\Theta(n)$ words. This space can become problematic for large n. This problem was perceived by Hon et al. [26], who proposed a data structure consisting of the input string T and two integer arrays, each of length n. Both arrays store, respectively, the beginning and the ending position of a SUS for each position i with $0 \leq i \leq n-1$. Hon et al. provided an algorithm that can construct these two arrays in linear time with $O(\log n)$ bits of additional working space, given that both arrays are stored in $2n \log n$ bits and that $\sigma \leq n$. Instead of building a data structure, Ganguly et al. [21] proposed a time-space trade-off algorithm using $O(n/\tau)$ words of additional working space, answering a given query in $O(n\tau^2 \log \frac{n}{\tau})$ time directly, for a trade-off parameter $\tau \geq 1$. They also proposed the first *compact* data structure of size $4n+o(n)$ bits that can answer a query in constant time. They can construct this data structure in $O(n \log n)$ time using $O(n \log \sigma)$ bits of additional working space.

Interval SUS Problems Hu et al. [27] were the first to consider the interval SUS problem. They proposed a data structure answering a query returning all SUSs for the respective query interval in $O(k)$ optimal time after $O(n)$ time preprocessing.

Minimal Unique Substrings A unique substring u of T is said to be a *minimal unique substring* (*MUS*) of T if any proper substring of u is not a unique substring. Ilie and Smyth [29] formalized MUSs and proposed a linear time algorithm to compute all MUSs of a given string T. MUSs has been heavily utilized for solving the SUS problems. Note that all the above algorithms for the SUS problems compute all MUSs of the given string (or some data structure which is essentially equivalent to MUSs) in the preprocessing.

1.2 Our Contributions

1.2.1 Combinatorial Properties on Unique Substrings

(A-1) Total Number of SUSs in String.

We show the tight upper bound $(3n-1)/2$ for the maximum number of substrings which can be a SUS for some text position in a string of length n. We also introduce the notion of *non-trivial* SUS and show an asymptotically-tight upper bound $2n$ of the number of non-trivial SUSs in a string. These are the first non-trivial results for combinatorial properties on the SUS problems.

(A-2) Relationship between Run-Length Encoded String and MUSs in String.

We focus on the well-known string compression method run-length encoding (RLE), and study relations between run-length encoded strings and unique substrings. We show that the number m of MUSs of a string is less than twice the size r of the string compressed with RLE. Furthermore, we show the upper bound is tight, that is, there exists a family of strings that satisfies $m = 2r - 1$.

1.2.2 Space-Efficient Data Structures for Computing Unique Substrings

(B-1) RLE Based Data Structure for Computing SUSs.

For the interval SUS problem, we propose a new data structure of size $O(r)$ that can answer any SUS query in $O(\sqrt{\log r/\log\log r} + k)$ time, where r is the RLE size of the input string T and k is the number of SUSs to output. This is based on the aforementioned result of the number of MUSs in RLE strings. All known methods for solving SUS problems require $O(n)$ words of space. We emphasize that the size $O(r)$ of our data structure is not worse than $O(n)$ since $r \leq n$ holds for any string, and thus, it can be sub-linear when the input string is well-compressible by RLE.

(B-2) Compact Data Structures for Computing SUSs.

We also give an alternative data structure of size $4n + o(n)$ bits which can answer any interval SUS query in optimal $O(k)$ time utilizing succinct data structures. Furthermore, we give a smaller data structure of size $2.6n + o(n)$ bits for the point SUS problem with the same query time. These $O(n)$ bits data structures can be constructed in $O(n)$ time, and the working space is also small.

1.2.3 Computing Unique Substrings in Semi-Dynamic Strings

For a string T, a unique substring u of T is called a *minimal unique palindromic substring* (*MUPS*) of T if u is a palindrome and any shorter palindromic substring of u is not unique in T. We treat the problems of maintaining MUSs and MUPSs in a semi-dynamic string, where we can append a character to the right-end of the string or delete the left-most character from the string. Note that our semi-dynamic setting is also known as the *sliding window model*; that is a type of stream processing where we are interested in information such as regularities or statistics within each window, not the whole string.

(C-1) Algorithms for Computing MUSs for a Semi-Dynamic String.

We propose an algorithm for maintaining MUSs in a semi-dynamic string running in amortized $O(\log \sigma)$ time per edit operation, using $O(n)$ space where σ is the size of the alphabet. For the offline setting, a linear-time algorithm for computing MUSs of a string over an integer alphabet are known. Our algorithm is the first result to compute them efficiently when both sides of a given string are dynamically changing.

(C-2) Algorithms for Computing MUPSs for a Semi-Dynamic String.

We propose an algorithm for maintaining MUPSs in a semi-dynamic string running in amortized $O(n \log \sigma)$ time per edit operation using $O(n)$ space. Similar to the case of MUSs, it is known that all MUPSs of a string over an integer alphabet can be computed offline in linear time. Also, our algorithm is the first result to compute them efficiently when both sides of a given string are dynamically changing. As a bonus, we also propose a sliding-window-algorithm for maintaining the data structure called palindromic tree (a.k.a. eertree) of a string that stores all distinct palindromes in the string.

1.3 Organization

The rest of this thesis is organized as follows. In Chapter 2, we give some notations and definitions. In Chapter 3, we study the maximum number of SUSs in a string. Then we show the tight upper bound for the point SUSs and an upper bound for the interval SUSs that is asymptotically tight. In Chapter 4, we tackle the SUS problems described above and propose a data structure of size linear to the size of RLE string. In Chapter 5, we again consider the SUS problems, and propose compact data structures of $O(n)$ bits of space. In Chapter 6, we consider the problem

of computing MUSs in semi-dynamic strings and propose an efficient algorithm. In Chapter 7, we consider the problem of computing MUPSs in semi-dynamic strings. For the sake of maintaining MUPSs, we propose an algorithm for maintaining the palindromic tree in semi-dynamic strings. Then, we show that we can maintain the set of MUPSs in a semi-dynamic string by applying the semi-dynamic algorithm for palindromic trees.

Chapter 2

Preliminaries

2.1 Notations

Strings. Let Σ be an *alphabet* of size σ. An element of Σ is called a *character*. An element of Σ^* is called a *string*. For $|\Sigma| = 2$, we call a string also a *bit array*. The length of a string T is denoted by $|T|$. The *empty string* ε is the string of length 0. For any $0 \leq i \leq |T| - 1$, $T[i]$ denotes the i-th character of T. If $T = xyz$, then x, y, and z are called a *prefix*, *substring*, and *suffix* of T, respectively. They are called a *proper* prefix, *proper* substring, and *proper* suffix of T if $x \neq T$, $y \neq T$, and $z \neq T$, respectively. If a non-empty string b is both a proper prefix and a proper suffix of T, then b is called a *border* of T. For any $0 \leq i \leq j \leq |T| - 1$, $T[i..j]$ denotes the substring of T starting at position i and ending at position j. For any $0 \leq i \leq |T| - 1$, $T[i..]$ denotes the suffix starting at position i, i.e., $T[i..] = T[i..|T| - 1]$. For convenience, let $T[i'..j'] = \varepsilon$ for any $i' > j'$. For a non-empty string w, the set of beginning positions of occurrences of w in T is denoted by $occ_T(w) = \{i \mid T[i..i + |w| - 1] = w\}$. Let $\#occ_T(w) = |occ_T(w)|$. For convenience, let $\#occ_T(\varepsilon) = |T| + 1$. For any strings X and Y, let $lcp(X, Y)$ denote the length of the *longest common prefix* of X and Y. For any string T and any $0 \leq i \leq j \leq |T| - 1$, let $lce_T(i, j)$ denote the *longest common extension* of i and j in T, i.e., $lce_T(i, j) = lcp(T[i..], T[j..])$. If a string X is lexicographically smaller than another string Y, then we write $X \prec Y$ or $Y \succ X$.

In what follows, we consider an arbitrarily fixed string T of length $n \geq 1$ over an alphabet Σ of size $\sigma \geq 2$.

Minimal/Shortest Unique Substrings. For any substring w of T, w is called *unique* in T if $\#occ_T(w) = 1$, *quasi-unique* in T if $1 \leq \#occ_T(w) \leq 2$, and *repeating* in T if $\#occ_T(w) \geq 2$.

Since every unique substring $u = T[i..j]$ of T occurs exactly once in T, we will sometimes identify u with its corresponding interval $[i, j]$. We also say that the interval $[i, j]$ is unique if the corresponding substring $T[i..j]$ is a unique substring of T. A unique substring $u = T[i..j]$ is said to be *right minimal unique* if for any $i \leq j' < j$, $T[i..j']$ is a repeat of T. A unique substring $u = T[i..j]$ is said to be *left minimal unique* if for any $i < i' \leq j$, $T[i'..j]$ is a repeat of T.

A substring $u = T[i..j]$ is said to be a *minimal unique substring* (*MUS*) of T if u is right minimal unique and left minimal unique. Let $\mathsf{MUS}_T = \{[i, j] \mid T[i..j] \text{ is a MUS of } T\}$ be the set of all *intervals* corresponding to the MUSs of T. From the definition of MUSs, the next lemma follows:

Lemma 2.1 ([29]). *No element of* MUS_T *is nested in another element of* MUS_T*, i.e., two different MUSs* $[i, j], [k, l] \in \mathsf{MUS}_T$ *satisfy* $[i, j] \not\subset [k, l]$ *and* $[k, l] \not\subset [i, j]$*. Therefore,* $0 < |\mathsf{MUS}_T| \leq |T|$.

For any substring $T[i..j]$ of T and an interval $[s, t] \subset [0, n-1]$, $T[i..j]$ is said to be a *shortest unique substring* (*SUS*) of T for interval $[s, t]$ if (1) $T[i..j]$ is a unique substring of T, (2) $[s, t] \subset [i, j]$, and (3) $T[i'..j']$ is a repeating substring of T for any i', j' with $[s, t] \subset [i', j']$ and $j' - i' < j - i$. In particular, a SUS for some interval $[p, p]$ of length 1 is said to be a SUS for position p and is sometimes referred to as a *point* SUS of T. Also, a SUS for some interval (including those of length 1) is sometimes referred to as an *interval* SUS in T. Given an interval $[s, t] \subset [0, n-1]$, $\mathsf{SUS}_T([s, t])$ denotes the set of interval SUSs of T for interval $[s, t]$. Also, given a text position $p \in [0, n-1]$, $\mathsf{SUS}_T(p)$ denotes the set of point SUSs of T for the point p. Given a query position $p \in [0, n-1]$ (resp. a query interval $[s, t] \subset [0, n-1]$), the *point* (resp. *interval*) *SUS problem* is to compute $\mathsf{SUS}_T(p)$ (resp. $\mathsf{SUS}_T([s, t])$). See Fig. 2.1 for an example depicting MUSs and SUSs.

Covers. For two intervals $[i, j]$ and $[x, y]$, let $cover([i, j], [x, y]) = [\min\{i, x\}, \max\{j, y\}]$ denote the shortest interval that contains the text positions i, j, x, and y. If the interval $[x, y]$ consists of a single point, i.e., $x = y$, $cover([i, j], [x, y])$ is denoted by $cover([i, j], x)$ when we want to emphasize on the fact that $x = y$.

Semi-Dynamic String and Sliding Window. In our *semi-dynamic* setting on strings, we can perform limited editing operations on the input string. Specifically, we can append a character to the right-end of the string, or delete the leftmost character from the string.

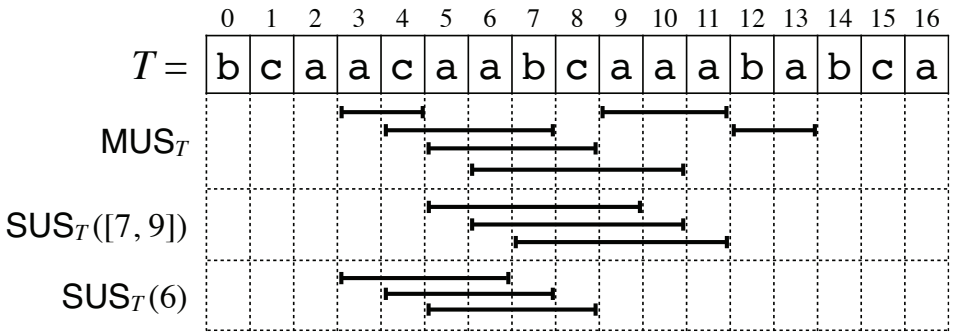

Figure 2.1: The string $T = \texttt{bcaacaabcaaababca}$ and the set $\mathsf{MUS}_T = \{[3,4], [4,7], [5,8], [6,10], [9,11], [12,13]\}$. MUS_T corresponds to the set $\{\texttt{ac}, \texttt{caab}, \texttt{aabc}, \texttt{abcaa}, \texttt{aaa}, \texttt{ba}\}$ of all MUSs of T. The substrings $T[5..9] = \texttt{aabca}$, $T[6..10] = \texttt{abcaa}$, and $T[7..11] = \texttt{bcaaa}$ are SUSs for the query interval $[7, 9]$. Also, the substrings $T[3..6] = \texttt{acaa}$, $T[4..7] = \texttt{caab}$, and $T[5..8] = \texttt{aabc}$ are SUSs for the query position 6.

Noticing that our semi-dynamic setting is essentially the same as the *sliding window model*, which is a type of streaming data processing. Because we will often refer to semi-dynamic strings as sliding windows over a long string T, we now formalize sliding windows over T. For each time $t = 0, 1, \ldots$, we consider the substring $T[i_t..j_t]$ called *the window at time* t. The windows must satisfy the following conditions: (1) $i_0 = j_0 = 0$ for the initial window at time 0; and (2) $0 \leq i_t \leq j_t \leq n-1$ and either $(i_t, j_t) = (i_{t-1}+1, j_{t-1})$ or $(i_t, j_t) = (i_{t-1}, j_{t-1}+1)$ for every time $t > 0$. In other words, the second condition means that we can either *delete* the leftmost character from the current window, or *append* a character to the right end of the current window at each time.

Given a sequence of windows (or equivalently, a sequence of delete/append operations), the aim of our sliding window problems is processing the windows in space proportional to the size of each window.

Model of Computation. Our model of computation is a standard word RAM with machine word size $\Omega(\log n)$.

2.2 Algorithmic Tools

In this section, we introduce data structures needed for our approach to compute unique substrings efficiently.

Range Minimum/Maximum Query. Given an integer array X of length n and an interval $[i, j] \subset [0, n-1]$, the *range minimum query* $\mathsf{RmQ}_X(i, j)$ (resp. the *range maximum query* $\mathsf{RMQ}_X(i, j)$) asks for the index p of a minimum element (resp. a maximum element) of the subarray $X[i..j]$, i.e., $p \in \arg\min_{i \le k \le j} X[k]$, or respectively $p \in \arg\max_{i \le k \le j} X[k]$. We use the following well-known data structure to handle these kind of queries:

Lemma 2.2 ([15])**.** *Given an integer array X of length n, there is an* RmQ *(resp.* RMQ*) data structure taking $2n + o(n)$ bits of space that can answer an* RmQ *(resp.* RMQ*) query on X in constant time. This data structure can be constructed in $O(n)$ time with $o(n)$ bits of additional working space.*

Rank/Select Query. Given a string Y of length n over the alphabet $[0, \sigma - 1]$. For an integer i with $0 \le i \le n-1$ and a character $c \in [0, \sigma - 1]$, the rank query $rank_Y(c, i)$ returns the number of the character c in the prefix $Y[0..i]$ of Y. Also, for an integer i with $1 \le i \le n$ and a character $c \in [0, \sigma - 1]$, the select query $select_Y(c, i)$ returns the position of Y containing the i-th occurrence of the character c (or returns the invalid symbol nil if such a position does not exist). For $\sigma = 2$ (i.e., Y is a bit array), we can make use of the following lemma:

Lemma 2.3 ([31, 11])**.** *We can endow a bit array Y of length n with a data structure answering $rank_Y$ and $select_Y$ in constant time. This data structure takes $o(n)$ bits of space, and can be built on Y in $O(n)$ time with $O(\log n)$ bits of additional working space.*

Predecessor/Successor Query. Let Z be an array of length m whose entries are non-negative integers in strictly increasing order. Further suppose that these integers are less than n. Given an integer d with $0 \le d \le n-1$, the *predecessor* and the *successor query* on Z with d are defined as $\mathsf{Pred}_Z(d) = \max\{i \mid Z[i] \le d\}$ and $\mathsf{Succ}_Z(d) = \min\{i \mid Z[i] \ge d\}$, where we stipulate that $\min\{\} = \max\{\} = nil$. There exists an $O(m)$ words data structure that can be built in $O(m\sqrt{\log m / \log\log m})$ time, such that later, for any given $0 \le d \le n-1$, $\mathsf{Pred}_Z(d)$ and $\mathsf{Succ}_Z(d)$ can be answered in $O(\sqrt{\log m / \log\log m})$ time [6].

Let BIT_Z be a bit array of length n marking all integers present in Z, i.e., $BIT_Z[i] = 1$ iff there is an integer j with $0 \le j \le m-1$ and $Z[j] = i$, for every i with $0 \le i \le n-1$. By endowing BIT_Z with a rank/select data structure, we yield an $n + o(n)$ bits data structure answering $\mathsf{Pred}_Z(d) = select_{BIT_Z}(1, rank_{BIT_Z}(1, d))$ and $\mathsf{Succ}_Z(d)$[1] in constant time for each d with $0 \le d \le n-1$.

[1] $\mathsf{Succ}_Z(d)$ can be computed similarly by considering the case whether $BIT_Z[d] = 1$.

Suffix Array and Related Arrays. We define six integer arrays $\mathsf{SA}_T[0..n-1]$, $\mathsf{ISA}_T[0..n-1]$, $\mathsf{LCP}_T[0..n]$, $\mathsf{PLCP}_T[0..n-1]$, $\mathsf{RankPred}_T[0..n-1]$ and $\mathsf{RankSucc}_T[0..n-1]$. The suffix array SA_T of T is the array with the property that $T[\mathsf{SA}_T[i]..]$ is lexicographically smaller than $T[\mathsf{SA}_T[i+1]..]$ for every i with $0 \leq i \leq n-2$ [43]. The inverse suffix array ISA_T of T is the inverse of SA_T, i.e., $\mathsf{SA}_T[\mathsf{ISA}_T[i]] = i$ for every i with $0 \leq i \leq n-1$. The longest common prefix array LCP_T of T is the array with the property that $\mathsf{LCP}_T[0] = \mathsf{LCP}_T[n] = 0$ and $\mathsf{LCP}_T[i] = lcp(T[\mathsf{SA}_T[i]..n], T[\mathsf{SA}_T[i-1]..n])$ for every i with $1 \leq i \leq n-1$. The permuted LCP array PLCP_T of T is the array storing the values of LCP_T in text position order (instead of suffix array order), i.e., $\mathsf{PLCP}_T[i] = \mathsf{LCP}_T[\mathsf{ISA}_T[i]]$ for every i with $0 \leq i \leq n-1$. The rank predecessor array $\mathsf{RankPred}_T$ of T is the array with the property that $\mathsf{RankPred}_T[\mathsf{SA}_T[0]] = nil$ and $\mathsf{RankPred}_T[\mathsf{SA}_T[i]] = \mathsf{SA}_T[i-1]$ for every i with $1 \leq i \leq n-1$. This array is also known as the Φ array in literature (e.g., [32, 22]). The rank successor array $\mathsf{RankSucc}_T$ of T is the array with the property that $\mathsf{RankSucc}_T[SA[n-1]] = nil$ and $\mathsf{RankSucc}_T[SA_T[i]] = SA_T[i+1]$ for every i with $0 \leq i \leq n-2$.

Suffix Tree. The *suffix tree* of string T, denoted $\mathsf{STree}(T)$, is a *compacted trie* that represents all suffixes of T. We consider a version of suffix trees a.k.a. *Ukkonen trees* [56]: Namely, $\mathsf{STree}(T)$ is a rooted tree such that

1. each edge is labeled by a non-empty substring of T,
2. each internal node has at least two children,
3. the out-going edges of each node begin with mutually distinct characters, and
4. the suffixes of T that are unique in T are represented by paths from the root to the leaves, and the other suffixes of T that are repeating in T are represented by paths from the root that end either on internal nodes or on edges.

To simplify the description of our algorithm, we assume that there is an auxiliary node $\perp$ which is the parent of only the root node. The out-going edge of $\perp$ is labeled with Σ; This means that we can go down from $\perp$ by reading any character in Σ. See Fig. 2.2 for an example of $\mathsf{STree}(T)$.

For each node v in $\mathsf{STree}(T)$, $parent(v)$ denotes the parent of v, $str(v)$ denotes the path string from the root to v, $depth(v)$ denotes the *string depth* of v (i.e., $depth(v) = |str(v)|$), and $subtree(v)$ denotes the subtree of $\mathsf{STree}(T)$ rooted at v. For each leaf ℓ in $\mathsf{STree}(T)$, $start(\ell)$ denotes the starting position of $str(\ell)$ in T. For each non-empty substring w of T, $hed(w) = v$

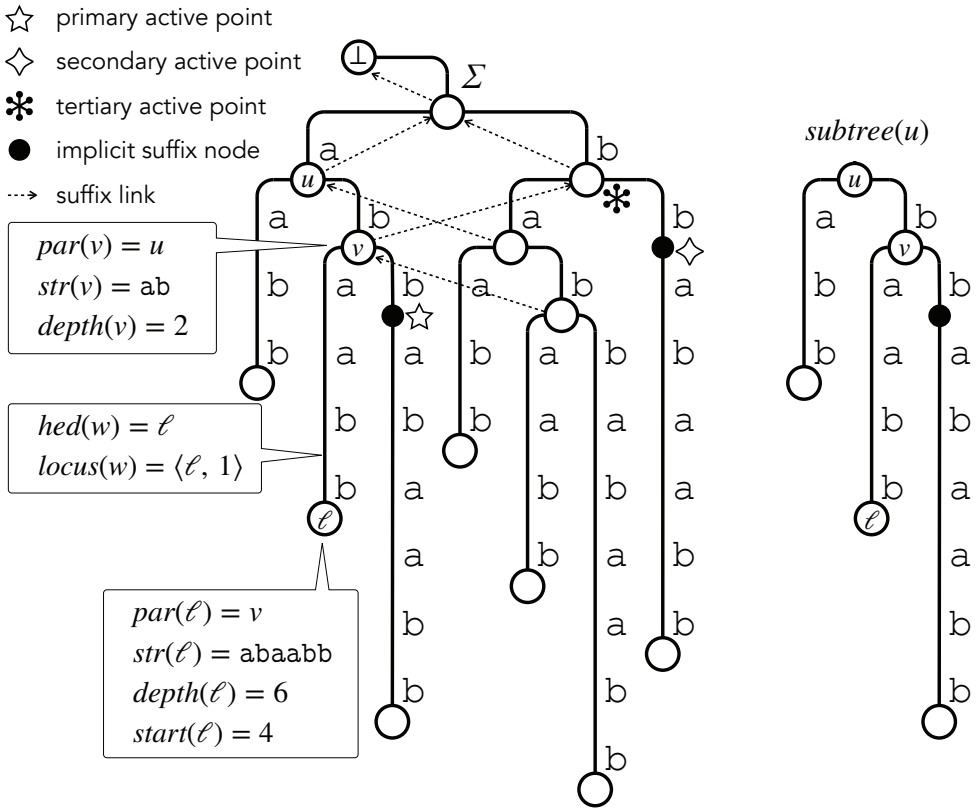

Figure 2.2: The suffix tree of string $T =$ `babbabaabb`, where the suffix links are depicted by broken arrows, the implicit suffix nodes are depicted by black circles, as well as the three kinds of active points, which will be defined in Chapter 6, are marked. For example of other notions on the suffix tree, substring $w =$ `abaab` of T is considered here.

denotes the *highest explicit descendant* where w is a prefix of $str(v)$ and $depth(parent(v)) < |w| \leq depth(v)$. For each substring w of T, $locus(w) = \langle u, h \rangle$ represents the locus in $\mathsf{STree}(T)$ where the path that spells out w from the root terminates, such that $u = hed(w)$ and $h = depth(u) - |w| \geq 0$. Namely, h is the off-set length from the child u of the locus for w when w is on an edge, and $h = 0$ when w is on a node (namely u). We say that a substring w of T with $locus(w) = \langle u, h \rangle$ is represented by an *explicit node* if $h = 0$, and by an *implicit node* if $h \geq 1$. We remark that in the Ukkonen tree $\mathsf{STree}(T)$ of a string T, some repeating suffixes may be represented by implicit nodes. An implicit node which represents a suffix of T is called an *implicit suffix node*. For any internal node v except for the root, the *suffix link* of v is a reversed edge from v to the explicit node that represents $str(v)[1..]$. The suffix link of the root that represents ε points to $\perp$.

Chapter 3

Tight Bounds on the Maximum Number of Shortest Unique Substrings

As we showed in Chapter 1, there are many algorithmic results on the SUS problems, however, structural properties of SUSs are not well understood. A trivial upper bound for the maximum number of intervals that correspond to point SUSs is $3n$, since every MUS can be a SUS for some position of the input string T, and for each query position p ($1 \leq p \leq n$), there can be at most 2 SUSs that are not MUSs (one that ends at position p and the other that begins at position p).

The main contribution of this chapter is matching upper and lower bounds for the maximum number of SUSs for the point SUS problem, which translate to "less than $1.5n$ point SUSs". Namely, we prove that any string of length n contains at most $(3n-1)/2$ SUSs for the point SUS problem. We give a series of strings which contains $(3n-1)/2$ SUSs for any odd number $n \geq 5$. Therefore, our bound is tight, and to our knowledge, this is the first non-trivial result for structural properties of SUSs. We also consider the maximum number of SUSs for the interval SUS problem. In so doing, we exclude a special case where a query interval $[s, t]$ itself is a unique substring that occurs exactly once in T. This is because we have $\Theta(n^2)$ bounds for such trivial SUSs. We then prove that any string of length n contains less than $2n$ *non-trivial* SUSs for the interval SUS problem. We also prove that there exists a string of length n which contains $(2-\varepsilon)n$ non-trivial SUSs for any small number $\varepsilon > 0$.

3.1 Preliminaries

Clearly, if $[i, j]$ is unique, then $[i, j]$ is the only SUS for the interval $[i, j]$. For any interval $[i, j]$ with $i < j$, if $[i, j]$ is unique and there is no other interval $[s, t] \subset [i, j]$ for which $[i, j]$ is a SUS, then we say that $[i, j]$ is a *trivial* interval SUS. Also, we say that $[i, j]$ is a *non-trivial* interval SUS if $[i, j]$ is not a trivial SUS.

For any interval $[s, t] \subset [0, |T| - 1]$, let $\mathsf{SUS}_T([s, t])$ denote the set of interval SUSs of T that contain query interval $[s, t]$, and $\mathcal{IS}_T$ the set of all non-trivial interval SUSs of T. Also, for any position $p \in [0, |T| - 1]$, let $\mathcal{PS}_T$ denote the set of all point SUSs of T, namely, $\mathcal{PS}_T = \bigcup_{p=0}^{n-1} \mathsf{SUS}_T(p)$.

3.2 Bounds on the Number of Point SUSs

Here we show a tight bound for the maximum number of point SUSs in a string. In this section, whenever we speak of SUSs, we mean point SUSs (those for the point SUS problem).

3.2.1 Upper Bound A

In this subsection, we show our first upper bound on the number of SUSs in a string T. In so doing, we define the subsets $\mathcal{LS}_T$, $\mathcal{MS}_T$, and $\mathcal{RS}_T$ of the set $\mathcal{PS}_T$ of all SUSs of string T by

$$\begin{aligned}
\mathcal{LS}_T &= \mathcal{PS}_T \cap \{[x, y] \notin \mathsf{MUS}_T \mid x < \exists i \leq y \; [i, y] \in \mathsf{MUS}_T\}, \\
\mathcal{MS}_T &= \mathcal{PS}_T \cap \mathsf{MUS}_T, \text{ and} \\
\mathcal{RS}_T &= \mathcal{PS}_T \cap \{[x, y] \notin \mathsf{MUS}_T \mid x \leq \exists j < y \; [x, j] \in \mathsf{MUS}_T\}.
\end{aligned}$$

Intuitively, $\mathcal{LS}_T$ is the set of SUSs of T which are *not* MUSs of T and can be obtained by extending the beginning positions of some MUSs to the left up to query positions, $\mathcal{MS}_T$ is the set of SUSs of T which are also MUSs of T, and $\mathcal{RS}_T$ is the set of SUSs of T which are *not* MUSs of T and can be obtained by extending the ending positions of some MUSs to the right up to query positions. It follows from their definitions that $\mathcal{LS}_T \cap \mathcal{MS}_T = \phi$, $\mathcal{MS}_T \cap \mathcal{RS}_T = \phi$, $\mathcal{RS}_T \cap \mathcal{LS}_T = \phi$ and that $\mathcal{PS}_T = \mathcal{LS}_T \cup \mathcal{MS}_T \cup \mathcal{RS}_T$. Fig. 3.2 in the next subsection shows examples of $\mathcal{LS}_T$, $\mathcal{MS}_T$, and $\mathcal{RS}_T$ for string $T = \texttt{aabbaababaa}$.

In the proof of the following theorem, we will evaluate the sizes of these three sets $\mathcal{LS}_T$, $\mathcal{MS}_T$, and $\mathcal{RS}_T$ separately.

Theorem 3.1. *For any string T, $|\mathcal{PS}_T| \leq 2|T| - |\mathsf{MUS}_T|$.*

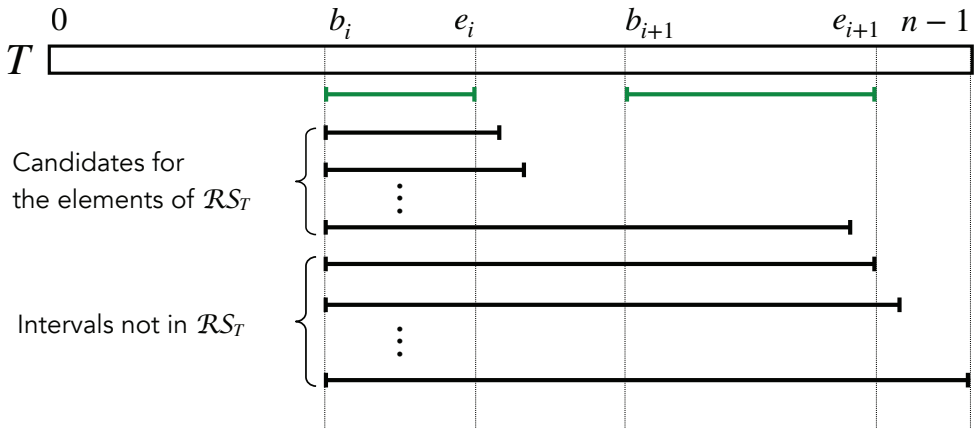

Figure 3.1: Illustration for Theorem 3.1. Consider two adjacent MUSs $[b_i, e_i]$ and $[b_{i+1}, e_{i+1}]$ depicted as the two intervals on the top. For any $e_i < e < e_{i+1}$, $[b_i, e]$ can be an element of $\mathcal{RS}_T$. On the other hand, for any $e' \geq e_{i+1}$, $[b_i, e']$ can never be an element of $\mathcal{PS}_T$ since $[b_i, e']$ contains two distinct MUSs $[b_i, e_i]$ and $[b_i, e_{i+1}]$, and hence $[b_i, e']$ can never be an element of $\mathcal{RS}_T$ as well.

Proof. Let $n = |T|$ and $m = |\mathsf{MUS}_T|$. For any $0 \leq i \leq m-1$, let $[b_i, e_i]$ denote the MUS of T that has the i-th smallest beginning position in MUS_T.

It is clear that $|\mathcal{MS}_T| \leq m$. Note that the inequality is due to that fact that some MUS may not be a point SUS for any position in T (such a MUS is called *meaningless* in the literature [55]).

Next, we consider the size of $\mathcal{RS}_T$. By definition, for any $[x, y] \in \mathcal{RS}_T$, x is equal to the beginning position of a MUS of T. Therefore, we can bound $|\mathcal{RS}_T|$ by summing up the number of SUSs that begin with b_i for every $[b_i, e_i] \in \mathsf{MUS}_T$. For any $0 \leq i \leq m-2$, consider two adjacent MUSs $[b_i, e_i], [b_{i+1}, e_{i+1}] \in \mathsf{MUS}_T$. Recall that $b_i < b_{i+1}$. Then, for any $j \geq e_{i+1}$, the interval $[b_i, j]$ contains both MUSs $[b_i, e_i]$ and $[b_{i+1}, e_{i+1}]$. This implies that $[b_i, j] \notin \mathcal{PS}_T$ (see Fig. 3.1), since otherwise both $[b_i, j]$ and $[b_{i+1}, j]$ are SUSs for position j, a contradiction. Thus, for any $[b_i, e_i] \in \mathsf{MUS}_T$ with $0 \leq i \leq m-2$, the number of SUSs that begin with b_i and belong to $\mathcal{RS}_T$ is at most $e_{i+1} - e_i - 1$. Also, the number of SUSs that begin with b_m and belong to $\mathcal{RS}_T$ is at most $n - e_m$. Consequently, we get $|\mathcal{RS}_T| = \sum_{i=0}^{m-2}(e_{i+1} - e_i - 1) + n - 1 - e_{m-1} = e_{m-1} - e_0 - (m-1) + n - 1 - e_{m-1} \leq n - m$.

A symmetric argument gives us the same bound for $|\mathcal{LS}_T|$, namely, $|\mathcal{LS}_T| \leq n - m$. Overall, we obtain $|\mathcal{PS}_T| = |\mathcal{LS}_T| + |\mathcal{MS}_T| + |\mathcal{RS}_T| \leq 2(n-m) + m = 2n - m$. $\square$

3.2.2 Upper Bound B

In this subsection, we provide another upper bound on the size of $\mathcal{PS}_T$.

Theorem 3.2. *For any string T, $|\mathcal{PS}_T| \leq |T| + |\mathsf{MUS}_T| - 1$.*

In order to show Theorem 3.2, we will use a function $f : \mathcal{PS}_T \to \{0, 1, \ldots, n-1\}$ and its inverse image $f^{-1} : \{0, 1, \ldots, n-1\} \to 2^{\mathcal{PS}_T}$. The next lemma is useful to define f and f^{-1}.

Lemma 3.1. *For any string T and interval $[x, y]$ such that $0 \leq x \leq y \leq |T|-1$, if $[x, y] \in \mathcal{RS}_T$ then $[x, y] \in \mathsf{SUS}_T(y)$, and if $[x, y] \in \mathcal{LS}_T$ then $[x, y] \in \mathsf{SUS}_T(x)$.*

Proof. We first prove the former case. Assume on the contrary that some $[x, y] \in \mathcal{RS}_T$ satisfies $[x, y] \notin \mathsf{SUS}_T(y)$. This implies that there exists a position p in T such that $x \leq p < y$ and $[x, y] \in \mathsf{SUS}_T(p)$. In addition, since $[x, y] \in \mathcal{RS}_T$, there exists a position q such that $x \leq q < y$ and $[x, q] \in \mathsf{MUS}_T$. Let $z = \max\{p, q\}$. Then, $T[x..z]$ is a unique substring of T which is shorter than $T[x..y]$ and contains position p. However, this contradicts that $T[x..y]$ is a SUS for position p. Thus, if $[x, y] \in \mathcal{RS}_T$ then $[x, y] \in \mathsf{SUS}_T(y)$. The latter case is symmetric and thus can be shown similarly. □

We are now ready to define f:

$$f([x, y]) = \begin{cases} x & \text{if } [x, y] \in \mathcal{LS}_T \cup \mathcal{MS}_T, \\ y & \text{if } [x, y] \in \mathcal{RS}_T. \end{cases}$$

Intuitively, the function f charges a given interval $[x, y]$ to its beginning position x if $[x, y]$ is an element of $\mathsf{MUS}_T \cap \mathcal{PS}_T$ or if $[x, y]$ is an element of $\mathsf{SUS}_T(p)$ for some query position p which is obtained by extending the left-end of a MUS to the left up to p. On the other hand, it charges $[x, y]$ to its ending position y if the interval is an element of $\mathsf{SUS}_T(p)$ for some query position p which is obtained by extending the right-end of a MUS to the right up to p. Fig. 3.2 shows examples for how the function f charges given interval $[x, y] \in \mathcal{PS}_T$.

We also define the inverse image f^{-1} of f as follows:

$$f^{-1}(u) = \{[x, y] \in \mathcal{PS}_T \mid f([x, y]) = u\}.$$

For positions u for which there is no element $[x, y]$ in $\mathcal{PS}_T$ satisfying $f([x, y]) = u$, let $f^{-1}(u) = \emptyset$. See also Fig. 3.2 for examples of f^{-1}.

By the definition of f^{-1}, it is clear that $|\mathcal{PS}_T| = \sum_{u=0}^{|T|-1} |f^{-1}(u)|$. Hence, in what follows we analyze $|f^{-1}(u)|$ for all positions u in string T.

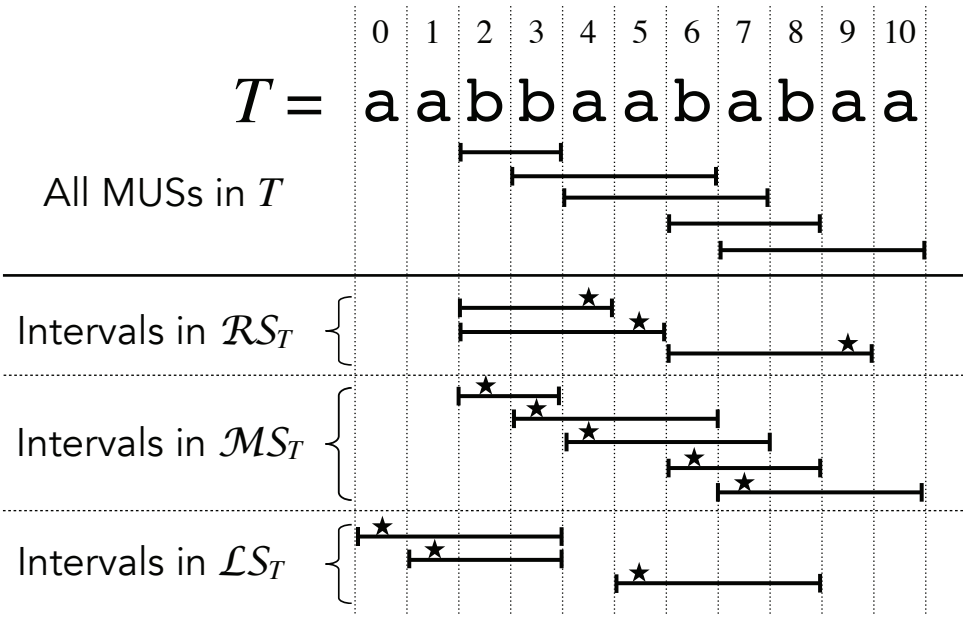

Figure 3.2: Illustration for functions f and f^{-1} of string $T = \texttt{aabbaababaa}$. The upper part of this diagram shows all MUSs in T, and the lower part shows all SUSs for all positions in T. Each star shows the position to which the function f maps the corresponding interval. Here, $\mathcal{RS}_T = \{[2,4],[2,5],[6,9]\}$, $\mathcal{MS}_T = \{[2,3],[3,6],[4,7],[6,8],[7,10]\}$, and $\mathcal{LS}_T = \{[0,3],[1,3],[5,8]\}$. Hence, we have $f([2,4]) = 4$, $f([2,5]) = 5$, $f([6,9]) = 9$, $f([2,3]) = 2$, $f([3,6]) = 3$, $f([4,7]) = 4$, $f([6,8]) = 6$, $f([7,10]) = 7$, $f([0,3]) = 0$, $f([1,3]) = 1$, and $f([5,8]) = 5$. For the inverse image, f^{-1}, we have $f^{-1}(0) = \{[0,3]\}$, $f^{-1}(1) = \{[1,3]\}$, $f^{-1}(2) = \{[2,3]\}$, $f^{-1}(3) = \{[3,6]\}$, $f^{-1}(4) = \{[2,4],[4,7]\}$, $f^{-1}(5) = \{[2,5],[5,8]\}$, $f^{-1}(6) = \{[6,8]\}$, $f^{-1}(7) = \{[7,10]\}$, $f^{-1}(8) = f^{-1}(10) = \emptyset$, and $f^{-1}(9) = \{[6,9]\}$.

Lemma 3.2. *For any string and position* $0 \leq u \leq |T| - 1$, $|f^{-1}(u)| \leq 2$.

Proof. Assume on the contrary that $|f^{-1}(u)| \geq 3$ for some position u in T. Let $[x_1, y_1]$, $[x_2, y_2]$ be any distinct elements of $f^{-1}(u)$. We firstly consider the following cases.

(1) Case where $[x_1, y_1], [x_2, y_2] \in \mathcal{LS}_T$: It follows from the definition of f^{-1} that $f([x_1, y_1]) = f([x_2, y_2]) = u$, and it follows from the definition of f that $x_1 = x_2 = u$. Since $[x_1, y_1]$ and $[x_2, y_2]$ are distinct, $y_1 \neq y_2$. Assume w.l.o.g. that $y_1 < y_2$. Then, $[x_2, y_2] = [u, y_2]$ is a SUS for position u but it is longer than another SUS $[x_1, y_1] = [u, y_1]$ for position u, a contradiction.

(2) Case where $[x_1, y_1], [x_2, y_2] \in \mathcal{MS}_T$: It follows from the definition of f^{-1} that $f([x_1, y_1]) = f([x_2, y_2]) = u$, and it follows from the definition of f that $x_1 = x_2 = u$. Since $[x_1, y_1]$

and $[x_2, y_2]$ are distinct, $y_1 \neq y_2$. Assume w.l.o.g. that $y_1 < y_2$. Then, $[x_2, y_2] = [u, y_2]$ is a MUS, but it contains another MUS $[x_1, y_1] = [u, y_1]$, a contradiction.

(3) Case where $[x_1, y_1], [x_2, y_2] \in \mathcal{RS}_T$: This is symmetric to Case (1) and thus we can obtain a contradiction in a similar way.

Hence, none of the above three cases is possible, and thus the remaining possibility is the case where $|f^{-1}(u)| = 3$ and each element of $f^{-1}(u)$ belongs to a different subset of $\mathcal{PS}_T$, namely, $f^{-1}(u) = \{[x_1, y_1], [x_2, y_2], [x_3, y_3]\}$ for some $[x_1, y_1] \in \mathcal{LS}_T$, $[x_2, y_2] \in \mathcal{MS}_T$, and $[x_3, y_3] \in \mathcal{RS}_T$. It follows from the definition of f^{-1} that $f([x_1, y_1]) = f([x_2, y_2]) = u$, and it follows from the definition of f that $x_1 = x_2 = u$. Since $[x_1, y_1]$ and $[x_2, y_2]$ are distinct, $y_1 \neq y_2$. There are two sub-cases.

(i) If $y_1 < y_2$, then a MUS $[x_2, y_2] = [u, y_2]$ contains a shorter SUS $[x_1, y_1] = [u, y_1]$ for position u, a contradiction.

(ii) If $y_1 > y_2$, then a SUS $[x_1, y_1] = [u, y_1]$ for position u contains a shorter MUS $[x_2, y_2] = [u, y_2]$, a contradiction.

Hence, neither of the sub-cases is possible.

Overall, we conclude that $|f^{-1}(u)| \leq 2$. □

By Lemma 3.2, for any position u in string T we have $|f^{-1}(u)| \leq 2$. Now let us consider any position u for which $|f^{-1}(u)| = 2$. We have the next lemma.

Lemma 3.3. *For any position u in string T for which $|f^{-1}(u)| = 2$, let $f^{-1}(u) = \{[x_1, y_1], [x_2, y_2]\}$ and assume w.l.o.g. that $x_1 \leq x_2$. Then, $x_1 \neq x_2$, $[x_1, y_1] \in \mathcal{RS}_T$ and $[x_2, y_2] \in \mathcal{LS}_T \cup \mathcal{MS}_T$.*

Proof. Suppose $x_1 = x_2$ and assume w.l.o.g. that $y_1 < y_2$. Then, from the definition of f, we have that ($x_1 = u$ or $y_1 = u$) and ($x_2 = u$ or $y_2 = u$) and thus $x_1 = x_2 = u$. Since $[x_2, y_2] \in f^{-1}(u)$ is not a MUS since it includes $[x_1, y_1]$, it must be that $[x_2, y_2] \in \mathsf{SUS}_T(u)$. This is a contradiction, because there exists a shorter unique substring $[x_1, y_1]$ that contains u. Thus we have $x_1 \neq x_2$. Assume on the contrary that $[x_1, y_1] \in \mathcal{LS}_T \cup \mathcal{MS}_T$. Then, it follows from the definition of f that $f([x_1, y_1]) = x_1$. In addition, since $[x_1, y_1] \in f^{-1}(u)$, we have $u = x_1$. This implies that $u = x_1 < x_2$, but it contradicts that $[x_2, y_2] \in f^{-1}(u)$. Thus, $[x_1, y_1] \notin \mathcal{LS}_T \cup \mathcal{MS}_T$, namely, $[x_1, y_1] \in \mathcal{RS}_T$. Now, it follows from the arguments in the proof of Lemma 3.2 that $[x_2, y_2] \notin \mathcal{RS}_T$, and hence $[x_2, y_2] \in \mathcal{MS}_T \cup \mathcal{LS}_T$. □

Let $m = |\mathsf{MUS}_T|$, and $\mathsf{MUS}_T = \{[b_1, e_1], \ldots, [b_m, e_m]\}$. The next corollary immediately follows from Lemmas 3.1 and 3.3.

Corollary 3.1. *For any position u in string T with $|f^{-1}(u)| = 2$, there exist two integers $1 \leq i < j \leq m$ such that $\mathsf{SUS}_T(u) = \{[b_i, u], [u, e_j]\}$.*

For any position u in string T before b_1 or after b_m, we have the next lemma.

Lemma 3.4. *For any position u in string T s.t. $0 \leq u \leq b_1$ or $b_m < u \leq n-1$, $|f^{-1}(u)| \leq 1$.*

Proof. Assume on the contrary that $|f^{-1}(u)| = 2$ for some $0 \leq u \leq b_1$. By Lemma 3.3, there exists $[x, y] \in f^{-1}(u)$ such that $[x, y] \in \mathcal{RS}_T$. By the definitions of f and f^{-1}, we have $y = u$. Also, by the definition of $\mathcal{RS}_T$, there exists a position $e < y$ in T such that $[x, e] \in \mathsf{MUS}_T$. Now we have $x \leq e < y = u \leq b_1$, however, this contradicts that b_1 is the beginning position of the first (leftmost) MUS in MUS_T. Thus $|f^{-1}(u)| \leq 1$ for any $0 \leq u \leq b_1$.

Assume on the contrary that $|f^{-1}(u)| = 2$ for some $b_m < u \leq n-1$. By Lemma 3.3, there exists $[x', y'] \in f^{-1}(u)$ such that $[x', y'] \in \mathcal{MS}_T \cup \mathcal{LS}_T$. By the definition of f and f^{-1}, we have $x' = u$. There are two cases to consider:

- If $[x', y'] \in \mathcal{MS}_T$, then $[x', y'] \in \mathsf{MUS}_T$. Thus $x' = u > b_m$ is the beginning position of a MUS in MUS_T, however, this contradicts that b_m is the beginning position of the last (rightmost) MUS in MUS_T.

- If $[x', y'] \in \mathcal{LS}_T$, then by the definition of $\mathcal{LS}_T$ there exists a position $b > x'$ such that $[b, y'] \in \mathsf{MUS}_T$. Now we have $b > x' = u > b_m$, however, this contradicts that b_m is the beginning position of the last (rightmost) MUS in MUS_T.

Consequently, $|f^{-1}(u)| \leq 1$ for any $b_m < u \leq n-1$. □

Lemma 3.5. *For any non-empty string T, let $U = \{u \mid |f^{-1}(u)| = 2\}$. Then, $|U| \leq |\mathsf{MUS}_T| - 1$.*

Proof. Let $n = |T|$ and $m = |\mathsf{MUS}_T|$. Recall that for any $0 \leq i \leq m-1$, $[b_i, e_i]$ denotes the i-th element of MUS_T.

Let $B = \{b_i \mid 0 \leq i \leq m-2\}$. We define function $g : U \to B$ as $g(u) = \max\{b < u \mid b \in B\}$. By the definition of U and Lemma 3.4, any position $u \in U$ satisfies $b_0 < u \leq b_{m-1}$. Therefore, $g(u)$ is well-defined for any position $u \in U$, and $g(u)$ returns the predecessor of u in the set B. It is clear that $|B| = m-1$. Thus, if g is an injection, then we immediately obtain the claimed bound $|U| \leq |B| = m-1$.

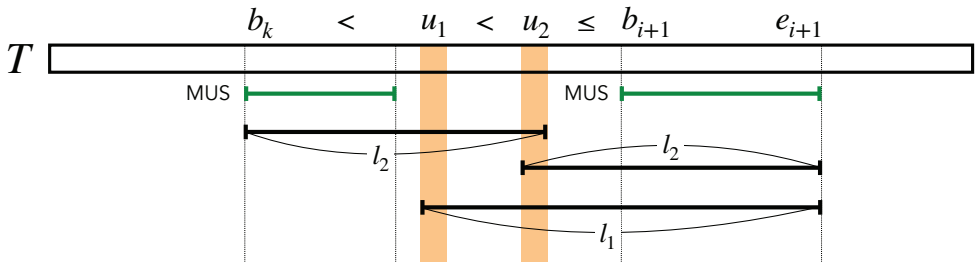

Figure 3.3: Illustration for Lemma 3.5. The two intervals show two MUSs $[b_k, e_k]$, $[b_{i+1}, e_{i+1}]$ $\in \mathsf{MUS}_T$, where $b_k \leq b_i$. Both $[b_k, u_2]$ and $[u_2, b_{i+1}]$ are SUSs for position u_2, and $[u_1, e_{i+1}]$ is a SUS for position u_1. Since $u_1 < u_2$, it holds that $l_1 > l_2$, where l_1 and l_2 are the lengths of SUSs for positions u_1 and u_2, respectively. Then, the interval $[b_k, u_2]$ of length l_2 contains position u_1 and $T[b_k..u_2]$ is a unique substring of T. However, this contradicts that l_1 is the length of each SUS for position u_1.

In what follows, we show that g is indeed an injection. Assume on the contrary that g is not an injection. Let u_1 and u_2 be elements in U such that $u_1 < u_2$ and $g(u_1) = g(u_2)$. Let $b_i \in B$ such that $b_i = g(u_1) = g(u_2)$. Then, by the definition of g, we have $b_i < u_1 < u_2 \leq b_{i+1}$. See Fig. 3.3 for illustration.

Let l_1 and l_2 be the lengths of the SUSs for positions u_1 and u_2, respectively. Since $|f^{-1}(u_2)| = 2$, it follows from Corollary 3.1 that there exists $b_k \in B$ such that $b_k \leq b_i$ and $\mathsf{SUS}_T(u_2) = \{[b_k, u_2], [u_2, e_{i+1}]\}$. This implies $l_2 = u_2 - b_k + 1 = e_{i+1} - u_2 + 1$. On the other hand, since $|f^{-1}(u_1)| = 2$, it follows from Corollary 3.1 that $[u_1, e_{i+1}] \in \mathsf{SUS}_T(u_1)$, which implies $l_1 = e_{i+1} - u_1 + 1$. Since $u_1 < u_2$, we have $l_1 > l_2$.

Now focus on a SUS $[b_k, u_2]$ for position u_2. Since $b_k \leq b_i < u_1 < u_2$, $[b_k, u_2]$ contains u_1. However, $[b_k, u_2]$ is a SUS for position u_2 and is of length $l_2 < l_1$. This contradicts that $[u_1, e_{i+1}]$ of length l_1 is each SUS for position u_1. Hence g is an injection. □

We are ready to prove the main result of this subsection, Theorem 3.2.

Proof. Let $n = |T|$, $m = |\mathsf{MUS}_T|$, $U = \{u \mid |f^{-1}(u)| = 2\}$, and $V = \{0, \ldots, n-1\} \setminus U$. It is clear that $|U| + |V| = n$. By Lemma 3.2, $V = \{u \mid |f^{-1}(u)| \leq 1\}$. Also, by Lemma 3.5, $|U| \leq m - 1$. Recall that $|\mathcal{PS}_T| = \sum_{u=1}^{n} |f^{-1}(u)|$. Putting all together, we obtain $|\mathcal{PS}_T| = \sum_{u=0}^{n-1} |f^{-1}(u)| \leq |V| + 2|U| = n + |U| \leq n + m - 1$. □

3.2.3 Matching Upper and Lower Bounds

We are ready to show the main result of this chapter.

Theorem 3.3. *For any non-empty string T, $|\mathcal{PS}_T| \leq (3|T|-1)/2$. This bound is tight, namely, for any odd $n \geq 5$ there exists a string T of length n s.t. $|\mathcal{PS}_T| = (3n-1)/2$.*

Proof. By Theorem 3.1, we have $|\mathsf{MUS}_T| \leq 2|T| - |\mathcal{PS}_T|$. Also, by Theorem 3.2, we have $|\mathcal{PS}_T| - |T| + 1 \leq |\mathsf{MUS}_T|$. Thus $|\mathcal{PS}_T| - |T| + 1 \leq 2|T| - |\mathcal{PS}_T|$, which immediately leads to the claimed bound $|\mathcal{PS}_T| \leq (3|T|-1)/2$.

We show that the above upper bound is indeed tight. For any odd number $n = 2k-1 \geq 5$, consider string $T = a_0 x a_1 x \cdots a_{k-2} x a_{k-1}$, where $a_0, \ldots, a_{k-1}, x \in \Sigma$, $a_i \neq a_j$ for all $0 \leq i \neq j \leq k-1$, and $x \neq a_i$ for all $0 \leq i \leq k-1$. For any $0 \leq i \leq k-1$, $T[2i] = a_i$ is a unique substring of T, and thus $[2i, 2i] \in \mathsf{SUS}_T(2i)$. Also, for any $0 \leq i \leq k-2$, $T[2i+1] = x$ is a repeating substring of T while $T[2i..2i+1] = a_i x$ and $T[2i+1..2i+2] = x a_{i+1}$ are unique substrings of T. This implies that $[2i, 2i+1], [2i+1, 2i+2] \in \mathsf{SUS}_T(2i+1)$. Hence, we have $|\mathcal{PS}_T| = k + 2(k-1) = 3k-2 = 3(n+1)/2 - 2 = (3n-1)/2$. □

3.2.4 Lower Bound for Fixed-Size Alphabet

The lower bound of Theorem 3.3 is due to a series of strings over an alphabet of unbounded size. In this subsection, we fix the alphabet size σ and present a series of strings that contain many point SUSs.

Theorem 3.4. *Let $n \geq 2$ and $2 \leq \sigma \leq (n+3)/2$. There exists a string T of length n over an alphabet of size σ such that $|\mathcal{PS}_T| = n + \sigma - 2$.*

Proof. Let $\Sigma = \{a_0, \ldots, a_{\sigma-2}, x\}$ and $T = a_0 x a_1 x \cdots a_{\sigma-2} x^{n-2\sigma+3}$. For any $0 \leq i \leq \sigma-2$, $T[2i] = a_i$ is a unique substring of T, and thus $[2i, 2i] \in \mathsf{SUS}_T(2i)$. For any $0 \leq j \leq \sigma-3$, $T[2j+1] = x$ is a repeating substring of T while $T[2j..2j+1] = a_j x$ and $T[2j+1..2j+2] = x a_{j+1}$ are unique substrings of T. This implies that $[2j, 2j+1], [2j+1, 2j+2] \in \mathsf{SUS}_T(2j+1)$. For any $2\sigma-3 \leq k \leq n-2$, $T[2\sigma-3..k] = x^{k-2\sigma+3}$ is a repeating substring of T while $T[2\sigma-4..k] = a_{\sigma-2} x^{k-2\sigma+3}$ is a unique substrings of T. This implies that $[2\sigma-4, k] \in \mathsf{SUS}_T(k)$. Also, $T[2\sigma-2..n-1] = x^{n-2\sigma+2}$ is a repeating substring of T and $T[2\sigma-3..n-1] = x^{n-2\sigma+3}$ is a unique substring of T, and thus $[2\sigma-3..n-1] \in \mathsf{SUS}_T(n-1)$. Summing up all the point SUSs above, we obtain $|\mathcal{PS}_T| = \sigma - 1 + 2(\sigma-2) + n - 2\sigma + 2 + 1 = n + \sigma - 2$. □

3.3 Bounds on the Number of Interval SUSs

In this section, we show almost tight bounds for the maximum number of non-trivial interval SUSs $\mathcal{IS}_T$ of a string T. The following upper bound for $|\mathcal{IS}_T|$ can be obtained in an analogous way to Theorem 3.1.

Lemma 3.6. *For any non-empty string T, $|\mathcal{IS}_T| \leq 2|T| - |\mathsf{MUS}_T|$.*

We also have the following lower bound for $|\mathcal{IS}_T|$.

Lemma 3.7. *For any $\varepsilon > 0$, there exists a string T of length n such that $|\mathcal{IS}_T| > (2-\varepsilon)n$.*

Proof. Let $x = \lceil 3/(2\varepsilon) \rceil$, $T = c_0 a^x c_1 a^x c_2$ and $n = |T| = 2x+3$. Clearly, c_0, c_1 and c_2 are MUSs of T and are in $\mathcal{IS}_T$. For all $1 \leq i \leq x$, $T[0..i]$ and $T[i..x+1]$ are unique substrings of T, and $T[1..i]$ and $T[i..x]$ are repeating substrings of T. This implies $T[0..i] \in \mathsf{SUS}_T([1,i])$ and $T[i..x+1] \in \mathsf{SUS}_T([i,x])$. Similarly, for all $x+2 \leq j \leq 2x+1$, $T[x+1..j] \in \mathsf{SUS}_T([x+2,j])$ and $T[j..2x+2] \in \mathsf{SUS}_T([j,2x+1])$. Then, we have $|\mathcal{IS}_T| = 4x+3$. Hence, $|\mathcal{IS}_T| - (2-\varepsilon)n = 4x+3-(2-\varepsilon)(2x+3) = 2\varepsilon x + 3\varepsilon - 3 = 2\varepsilon\lceil 3/(2\varepsilon)\rceil + 3\varepsilon - 3 \geq 3\varepsilon > 0$. □

As is shown in the following theorem, the number of non-trivial interval SUSs contained in the string T of Lemma 3.7 "almost coincides" with the upper bound of Lemma. Namely:

Theorem 3.5. *For any $\varepsilon > 0$, there is a string T such that $(2|T| - |\mathsf{MUS}_T|) - (2-\varepsilon)|T| \leq 5\varepsilon$.*

Proof. For any $\varepsilon > 0$, consider the string T of Lemma 3.7. We remark that T contains 3 MUSs, namely, $|\mathsf{MUS}_T| = 3$. Hence, we obtain $(2|T| - |\mathsf{MUS}_T|) - (2-\varepsilon)|T| = \varepsilon|T| - |\mathsf{MUS}_T| = \varepsilon|T| - 3 = \varepsilon(2\lceil 3/(2\varepsilon)\rceil + 3) - 3 = 2\varepsilon\lceil 3/(2\varepsilon)\rceil + 3\varepsilon - 3 \leq 2\varepsilon(3/(2\varepsilon)+1) + 3\varepsilon - 3 = 5\varepsilon \to 0\,(\varepsilon \to 0)$. □

3.4 Conclusions and Open Questions

In this chapter, we presented matching upper and lower bounds for the maximum number of SUSs for the point SUS problem. Namely, we proved that any string of length n can contain at most $(3n-1)/2$ SUSs for the point SUS problem, and showed that this bound is tight by giving a string of length n containing $(3n-1)/2$ SUSs. For a fixed alphabet size σ, we also presented a string of length n containing $n+\sigma-2$ SUSs. Moreover, we showed that any string of length n which contains m MUSs can have at most $2n-m$ non-trivial interval SUSs, and that for any $\varepsilon > 0$ there is a string of length n which contains $(2-\varepsilon)n$ non-trivial interval SUSs.

An interesting future work is to show a non-trivial upper bound of the maximum number of point SUSs for a fixed alphabet size σ. We conjecture that the tight upper bound matches our lower bound $n + \sigma - 2$. Another future work is to close the small gap between the upper and lower bounds on the maximum number of non-trivial interval SUSs shown in Theorem 3.5.

Chapter 4

Shortest Unique Substring Queries on Run-Length Encoded Strings

In this chapter, we consider the interval SUS problem in the case where the string is given in *run-length encoding* (*RLE*). String processing on the compressed representation of a string without explicit decompression [1] is a heavily studied topic, and can lead to time and space efficient processing [53]. There have been many studies on efficient algorithms for processing RLE strings [7, 8, 2, 3, 40, 34, 10, 39, 4]. We show that given a run-length encoding of size r of a string, we can construct a data structure of size $O(r + \pi_s(n, r))$ in $O(r \log r + \pi_c(n, r))$ time such that all SUSs that contain the query interval can be answered in $O(\pi_q(n, r) + k)$ time, where k is the number of such SUSs and $\pi_s(n, r)$, $\pi_c(n, r)$, $\pi_q(n, r)$ are, respectively, the size, construction time, and query time for a predecessor/successor query data structure of r elements for the universe of $[0, n - 1]$. Using the data structure by Beam and Fich [6], this results in a data structure of size $O(r)$ space that is constructed in $O(r \log r)$ time, and answers queries in $O(\sqrt{\log r / \log \log r} + k)$ time. Thus, compared to previous work [27], our algorithm allows for more time and space efficient preprocessing for RLE compressible strings, with a slight increase in query time.

Our result is an outcome of a non-trivial mixed use of combinatorial properties of RLE strings and data structures built on RLE strings: All existing solutions [49, 55, 28, 27, 26] to the SUS problem precompute *minimal unique substrings* (*MUSs*) of a given string, which are minimal substrings of T occurring exactly once in T, and store them in $\Theta(n)$ space, since, in general, there can be $\Theta(n)$ MUSs in a given string. However, using combinatorial properties of MUSs and RLE strings, we show in this chapter that any string of RLE size r contains at most $2r - 1$ MUSs, enabling our space-efficient $O(r)$-size data structure for the SUS problem.

This bound is indeed tight, namely, some strings contain $2r - 1$ MUSs. In our algorithm, we separately treat MUSs that are completely contained in runs, those that start at the last characters of runs, and the rest. We then show that all the MUSs can be precomputed in $O(r \log r)$ time using a special type of suffix arrays for RLE strings [54]. Finally, we show how, given all MUSs, to efficiently compute all SUSs for any given query interval.

4.1 Preliminaries

4.1.1 Run-Length Encoding.

The *run-length encoding* (*RLE*) of string T, denoted by $RLE(T)$, is a compact representation of T which encodes each maximal character run $T[i..i+e-1]$ by a^e, if (1) $T[j] = a$ for all $i \leq j \leq i+e-1$, (2) $T[i-1] \neq T[i]$ or $i = 0$, and (3) $T[i+e-1] \neq T[i+e]$ or $i+e-1 = n-1$. E.g., $RLE(\texttt{aabbbbcccaaa\$}) = \texttt{a}^2\texttt{b}^4\texttt{c}^3\texttt{a}^3\$^1$. The *size* of $RLE(T) = a_0^{e_0} \cdots a_{r-1}^{e_{r-1}}$ is the number r of maximal character runs in T and is denoted by $|RLE(T)|$. For any $0 \leq i \leq r-1$, let $bpos_T(i)$, $epos_T(i)$, and $exp_T(i)$ respectively denote the beginning position, ending position, and exponent of the i-th run of $RLE(T)$ in the original string T; namely, $bpos_T(i) = \sum_{k=0}^{i-1} e_k$, $epos_T(i) = \sum_{k=0}^{i} e_k - 1$, and $exp_T(i) = e_i$.

Sparse Suffix Array and Related Arrays for RLE Strings. Let $B \subseteq [0, n-1]$ be any subset of positions in T called sampled positions. The *sparse suffix array* SSA_B of a string T w.r.t. B is an array of size $|B|$ such that $\mathsf{SSA}_B[i] \in B$ for all $0 \leq i \leq |B|-1$ and $T[\mathsf{SSA}_B[i]..] \prec T[\mathsf{SSA}_B[i+1]..]$ for all $0 \leq i < n-1$.

Let $r = |RLE(T)|$ and $E = \{epos_T(i) \mid 0 \leq i \leq r-1\}$. The *truncated RLE suffix array* for $RLE(T)$, denoted tRLESA_T, is the sparse suffix array of T w.r.t. E. Namely, for any $0 \leq i \leq r-1$, $\mathsf{tRLESA}_T[i] = j$ iff $j \in E$ and the lexicographical rank of the suffix $T[j..]$ is i among all suffixes of T that begin with positions in E. Let tRLESA_T^{-1} be an array of size r such that $\mathsf{tRLESA}_T[\mathsf{tRLESA}_T^{-1}[i]] = epos_T(i)$ for all $0 \leq i \leq r-1$. Let $\mathsf{tRLELCP}_T$ be an array of size $r+1$ such that $\mathsf{tRLELCP}_T[0] = \mathsf{tRLELCP}_T[r] = 0$ and $\mathsf{tRLELCP}_T[i] = lce_T(\mathsf{tRLESA}_T[i-1], \mathsf{tRLESA}_T[i]) = lcp(T[\mathsf{tRLESA}_T[i-1]..], T[\mathsf{tRLESA}_T[i]..])$ for all $1 \leq i \leq r-1$. Also, let EXP_T be an array of size r such that $\mathsf{EXP}_T[i] = exp_T(k)$ where $\mathsf{tRLESA}_T[i] = epos_T(k)$ for all $0 \leq i \leq r-1$, namely, $\mathsf{EXP}_T[i]$ stores the ignored exponent of the first run of the i-th suffix in tRLESA_T. See Fig. 4.1 for concrete examples of these arrays.

	tRLESA_T^{-1}	EXP_T	tRLESA_T	$\mathsf{tRLELCP}_T$	
0	2	2	9	0	$\mathtt{a}^{(2)}\ \mathtt{b}^2\ \mathtt{c}^3\ \$^1$
1	5	1	5	1	$\mathtt{a}^{(1)}\ \mathtt{c}^2\ \mathtt{a}^2\ \mathtt{b}^2\ \mathtt{c}^3\ \$^1$
2	1	3	2	4	$\mathtt{a}^{(3)}\ \mathtt{c}^2\ \mathtt{a}^1\ \mathtt{c}^2\ \mathtt{a}^2\ \mathtt{b}^2\ \mathtt{c}^3\ \$^1$
3	4	2	11	0	$\mathtt{b}^{(2)}\ \mathtt{c}^3\ \$^1$
4	0	2	7	0	$\mathtt{c}^{(2)}\ \mathtt{a}^2\ \mathtt{b}^2\ \mathtt{c}^3\ \$^1$
5	3	2	4	2	$\mathtt{c}^{(2)}\ \mathtt{a}^1\ \mathtt{c}^2\ \mathtt{a}^2\ \mathtt{b}^2\ \mathtt{c}^3\ \$^1$
6	6	3	14	1	$\mathtt{c}^{(3)}\ \$^1$
7	7	1	15	0	$\$^{(1)}$
8				0	

Figure 4.1: tRLESA_T, tRLESA_T^{-1}, $\mathsf{tRLELCP}_T$, and EXP_T for $RLE(T) = \mathtt{a}^3\mathtt{c}^2\mathtt{a}^1\mathtt{c}^2\mathtt{a}^2\mathtt{b}^2\mathtt{c}^3\$^1$ with $r = 8$ and $n = |T| = 16$. We remark that the exponents of the first runs in parentheses are all regarded as 1. For instance, consider the suffixes of lexicographical ranks 1 and 2. Although $\mathtt{a}^1\mathtt{c}^2\mathtt{a}^2\mathtt{b}^2\mathtt{c}^3\$^1$ is lexicographically greater than $\mathtt{a}^3\mathtt{c}^2\mathtt{a}^1\mathtt{c}^2\mathtt{a}^2\mathtt{b}^2\mathtt{c}^3\$^1$, $\mathtt{a}^1\mathtt{c}^2\mathtt{a}^2\mathtt{b}^2\mathtt{c}^3\$^1$ is lexicographically smaller than $\mathtt{a}^1\mathtt{c}^2\mathtt{a}^1\mathtt{c}^2\mathtt{a}^2\mathtt{b}^2\mathtt{c}^3\$^1$, and tRLESA_T builds on the latter ordering.

Lemma 4.1 ([54])**.** *Given $RLE(T)$ of size r, tRLESA_T, tRLESA_T^{-1}, $\mathsf{tRLELCP}_T$, and EXP_T can be computed in a total of $O(r \log r)$ time with $O(r)$ working space.*

The following is a simple observation of these arrays we will exploit.

Observation 4.1. *For any $0 \leq i \leq r-1$, let $l = \max\{\mathsf{tRLELCP}_T[p], \mathsf{tRLELCP}_T[p+1]\}$, where $p = \mathsf{tRLESA}_T^{-1}[i]$. If $l \neq 0$, then l is the length of the longest repeat of T that starts at $epos_T(i)$.*

For example of Observation 4.1, see Fig. 4.1. There, for position $i = 2$, we have $p = \mathsf{tRLESA}_T^{-1}[2] = 1$. Then, observe that $l = \max\{1, 4\} = 4$ is the length of the longest repeat $\mathtt{ac}^2\mathtt{a}$ that starts at position $epos_T(2) = 5$. On the other hand, for position $i = 5$, we have $p = \mathsf{tRLESA}_T^{-1}[5] = 3$. Then, $l = \max\{0, 0\} = 0$, but this is not equal to the length 1 of the longest repeat $\mathtt{b}$ that starts at position $epos_T(5) = 11$. In our algorithm, we will use Observation 4.1 only the case where $l \neq 0$.

In this chapter, we will tackle the following problem:

Problem 4.1 (SUSs on RLE strings)**.**

Preprocess: $RLE(T) = a_0^{e_0} \cdots a_{r-1}^{e_{r-1}}$ *of size r of string T of length n.*

Query: *An interval $[s, t] \in [0, n-1]$.*

Return: *All SUSs of T containing the query interval $[s, t]$.*

4.1.2 Some Functions Related to tRLESA

In this subsection, we introduce some functions related to tRLESA_T and the other arrays, which will be used in our algorithm to compute SUSs on RLE strings.

Consider $RLE(T)$ of size r. For any pair $(i, j) \in [0, r-1] \times [0, r-1]$, let $trle\_lce_T(i, j) = lce_T(\mathsf{tRLESA}_T[i], \mathsf{tRLESA}_T[j])$. Since

$$trle\_lce_T(i, j) = \begin{cases} \mathsf{RmQ}_{\mathsf{tRLELCP}_T}(i+1, j) & \text{if } i < j, \\ \mathsf{RmQ}_{\mathsf{tRLELCP}_T}(j+1, i) & \text{otherwise,} \end{cases}$$

after a linear-time preprocessing on $\mathsf{tRLELCP}_T$, we can answer $trle\_lce_T(i, j)$ in $O(1)$ time for any given pair (i, j).

For any $0 \leq q \leq r-1$ and $e \geq 1$, let $exp\_pos(q, e)$ denote a query which returns a position $q' \neq q$, if it exists, that satisfies $\mathsf{EXP}_T[q'] \geq e$ and $T[\mathsf{tRLESA}_T[q']] = T[\mathsf{tRLESA}_T[q]]$ while maximizing $trle\_lce(q, q')$, and nil otherwise. Thus, with $q' = exp\_pos(q, e)$, we can obtain the length of the longest repeating substring starting at position $\mathsf{tRLESA}_T[q] - e + 1$ as $e - 1 + trle\_lce(q, q')$.

Lemma 4.2. *Given* EXP_T *for* $RLE(T)$ *of size* r*, we can preprocess* EXP_T *in* $O(r)$ *time so that subsequent* $exp\_pos(q, e)$ *queries can be answered in* $O(\log m)$ *time for any* $0 \leq q \leq m-1$ *and* $e \geq 1$.

Proof. We construct an RMQ data structure for EXP_T in $O(r)$ time. Since lexicographically close strings share a longer prefix, $exp\_pos(q, e)$ is one of the two closest neighbours of q in EXP_T that stores an exponent at least e, corresponding to a run of the same character. Thus, we can compute $exp\_pos(q, e)$ using two binary searches on EXP, by comparing e with the answer of the RMQ queries, starting with the initial range $[0, q-1]$ and $[q+1, r-1]$. Since the size of EXP_T is r and each RMQ query takes $O(1)$ time, it takes $O(\log r)$ time to locate $exp\_pos(q, e)$. □

For any $0 \leq q \leq r-1$ and $\ell \geq 0$, let $lce\_pos(q, \ell)$ denote a query which returns a position $q' \neq q$, if it exists, such that $trle\_lce(q, q') \geq \ell$ while maximizing $\mathsf{EXP}_T[q']$, and nil otherwise. In other words, $lce\_pos(q, \ell)$ corresponds to a suffix that has the maximum exponent out of suffixes which, have a common prefix of length ℓ with the suffix corresponding to q. Note that if $\ell > \max\{\mathsf{tRLELCP}[q], \mathsf{tRLELCP}[q+1]\}$, $lce\_pos(q, \ell) = nil$.

Lemma 4.3. *Given* $\mathsf{tRLELCP}_T$ *for* $RLE(T)$ *of size* r*, we can preprocess* $\mathsf{tRLELCP}_T$ *in* $O(r)$ *time so that subsequent* $lce\_pos(q, \ell)$ *queries can be answered in* $O(\log r)$ *time for any* $0 \leq q \leq r-1$ *and* $\ell \geq 0$.

Proof. We construct an RmQ data structure on $\mathsf{tRLELCP}_T$. Since, as noted previously, lexicographically close strings share a longer prefix, values of $trle\_lce(q, q'')$ are larger when q'' is closer to q. Thus, similar to Lemma 4.2, we can conduct two binary searches on $\mathsf{tRLELCP}_T$ using RmQ and obtain the maximal range $[q_p, q_n]$ such that $trle\_lce(q, q'') \geq \ell$ if and only if $q'' \in [q_p, q_n]$. After finding the range, the larger of the two RMQ queries for the ranges $[q_p, q-1]$ and $[q+1, q_n]$ on EXP_T gives the answer. □

4.2 Computing MUSs from RLE Strings

In this section we show how we can compute MUS_T given $RLE(T)$, which is the main part of our preprocessing. As will be seen in Section 4.2.1, we partition MUSs into three disjoint groups; those that are completely contained in runs, those that start at the last characters of runs, and the rest.

4.2.1 Size of MUS_T

We begin with the analysis of the size of MUS_T in terms of $r = |RLE(T)|$. Let

$$\begin{aligned}
\mathcal{M}^{(1)} =& \{[x, y] \in \mathsf{MUS}_T \mid bpos_T(i) \leq x \leq y \leq epos_T(i) \text{ for some } 0 \leq i \leq r-1\},\\
\mathcal{M}^{(2)} =& \{[x, y] \in \mathsf{MUS}_T \mid x = epos_T(i) < y \text{ for some } 0 \leq i < r-1\}, \text{ and}\\
\mathcal{M}^{(3)} =& \{[x, y] \in \mathsf{MUS}_T \mid bpos_T(i) \leq x < epos_T(i) < y \text{ for some } 0 \leq i < r-1\}.
\end{aligned}$$

Clearly, $\mathsf{MUS}_T = \mathcal{M}^{(1)} \cup \mathcal{M}^{(2)} \cup \mathcal{M}^{(3)}$. For example, for string $T = \texttt{aaaccaccaabbccc\$}$ as in Fig. 4.1, $\mathsf{MUS}_T = \{[0,2], [1,3], [4,6], [7,9], [9,10], [10,11], [11,12], [12,14], [15,15]\} = \{\texttt{aaa}, \texttt{aac}, \texttt{cac}, \texttt{caa}, \texttt{ab}, \texttt{bb}, \texttt{bc}, \texttt{ccc}, \$\}$, $\mathcal{M}^{(1)} = \{\texttt{aaa}, \texttt{bb}, \texttt{ccc}, \$\}$, $\mathcal{M}^{(2)} = \{\texttt{cac}, \texttt{caa}, \texttt{ab}, \texttt{bc}\}$, and $\mathcal{M}^{(3)} = \{\texttt{aac}\}$.

Since, by definition, a MUS cannot be a proper substring of another MUS, there can be at most one MUS that starts at any given position. Thus, it follows that $|\mathcal{M}^{(2)}| \leq r-1$.

For $|\mathcal{M}^{(3)}|$, we have the following lemma.

Lemma 4.4. *For any* $[x, y] \in \mathcal{M}^{(3)}$ *and* $p \in occ_T(T[x+1..y]) \setminus \{x+1\}$, *we have that* $p = bpos_T(i)$ *for some* $0 \leq i < r-1$.

Proof. Since $T[x..y]$ is a MUS, $T[x+1..y]$ is not unique and thus $occ_T(T[x+1..y]) \setminus \{x+1\}$ is not empty. If $p \neq bpos_T(i)$ for any $0 \leq i < r-1$, then $T[p-1] = T[p]$ and thus gives another occurrence of $T[x..y]$ contradicting that it is unique. □

We now show $|\mathcal{M}^{(3)} \cup \mathcal{M}^{(1)}| \leq r$. Let $\mathcal{R} = \{bpos_T(i) \mid 0 \leq i \leq r-1\}$. From Lemma 4.4, we can define a function $f : \mathcal{M}^{(3)} \cup \mathcal{M}^{(1)} \to \mathcal{R}$ as follows:

$$f([x,y]) = \begin{cases} \min(occ_T(T[x+1..y]) \setminus \{x+1\}) & \text{if } [x,y] \in \mathcal{M}^{(3)} \\ x & \text{if } [x,y] \in \mathcal{M}^{(1)} \end{cases}$$

Suppose f is not an injective function, i.e., there exist distinct intervals $[x_1, y_1], [x_2, y_2] \in \mathcal{M}^{(3)} \cup \mathcal{M}^{(1)}$ such that $x_1 \neq x_2$ and $p = f([x_1, y_1]) = f([x_2, y_2])$. Note that by definition, $\mathcal{M}^{(3)} \cap \mathcal{M}^{(1)} = \emptyset$.

If $[x_1, y_1], [x_2, y_2] \in \mathcal{M}^{(3)}$, assume w.l.o.g. $y_1 - x_1 \leq y_2 - x_2$. By definition of f, we have $T[x_1+1..y_1] = T[p..p+y_1-x_1-1]$ and $T[x_2+1..y_2] = T[p..p+y_2-x_2-1]$. Also, from the definition of $\mathcal{M}^{(3)}$, we have $T[x_1] = T[x_1+1] = T[p] = T[x_2+1] = T[x_2]$. It follows that $T[x_1..y_1]$ is a prefix of $T[x_2..y_2]$, contradicting that $T[x_1..y_1]$ is unique. If $[x_1, y_1] \in \mathcal{M}^{(3)}$ and $[x_2, y_2] \in \mathcal{M}^{(1)}$, this implies that $T[x_2..y_2]$ is a prefix of $T[x_1+1..y_1]$ which is not unique, thus contradicting that $T[x_2..y_2]$ is unique. Finally, if $[x_1, y_1], [x_2, y_2] \in \mathcal{M}^{(1)}$, $p = f([x_1, y_1]) = f([x_2, y_2])$ implies that $p = x_1 = x_2$ contradicting that $x_1 \neq x_2$. Thus, f must be an injective function. Therefore, $|\mathcal{M}^{(3)} \cup \mathcal{M}^{(1)}| \leq |\mathcal{R}| = r$.

From the above arguments, we have:

Lemma 4.5. $|\mathsf{MUS}_T| \leq 2r - 1$.

We note that the upper bound of Lemma 4.5 is tight, and there exists a string T such that $|\mathsf{MUS}_T| = 2r - 1$. Consider $T = a_0^{e_0} a_1^{e_1} \cdots a_{r-1}^{e_{r-1}}$ such that for any $0 \leq i, j \leq r-1$, $e_i \geq 2$, and $a_i \neq a_j$ when $i \neq j$. Clearly, $a_i^{e_i}$ is a MUS for all $0 \leq i \leq r-1$, and $a_i a_{i+1}$ is a MUS for all $0 \leq i < r-1$, giving $2r - 1$ MUSs.

4.2.2 Computing MUS_T

We now show how to obtain MUS_T in $O(r \log r)$ time and $O(r)$ space, by computing the sets $\mathcal{M}^{(1)}, \mathcal{M}^{(2)}, \mathcal{M}^{(3)}$ as defined in Section 4.2.1.

Computing $\mathcal{M}^{(1)}$

To compute $\mathcal{M}^{(1)}$, we first show a necessary and sufficient condition for an interval $[x, y]$ to be in $\mathcal{M}^{(1)}$.

Lemma 4.6. *For any string T where $RLE(T) = a_0^{e_0} \cdots a_{r-1}^{e_{r-1}}$, an interval $[x, y] \in \mathcal{M}^{(1)}$ if and only if there exists some $0 \leq i \leq r-1$ such that $bpos_T(i) = x$, $epos_T(i) = y$, and for any $j \in [0, r-1] \setminus \{i\}$, either $a_i \neq a_j$ or $e_j < e_i$.*

Proof. ($\Rightarrow$) Since $[x, y]$ is a MUS and any proper substring of $[x, y]$ is not unique, it must be that $x = bpos_T(i), y = epos_T(i)$ for some $0 \leq i \leq r - 1$. Furthermore, it must be that $a_i \neq a_j$ or $e_j < e_i$ for any $j \in [0, r - 1] \setminus \{i\}$, since otherwise, $[x, y]$ will not be unique. ($\Leftarrow$) The condition implies that $T[x..y]$ is the longest run of character a_i in T and is unique. Since any proper substring of $T[x..y]$ is not unique, $[x, y]$ is a MUS and is thus in $\mathcal{M}^{(1)}$. □

Let Σ_T be the subset of Σ consisting of letters occurring in T. Using Lemma 4.6, we can compute $\mathcal{M}^{(1)}$ by simply checking for each character $a \in \Sigma_T$, whether there exists a run of character a with a unique (w.r.t. runs of character a) maximum exponent, and if so, include the interval corresponding to the run in $\mathcal{M}^{(1)}$. Since $|\Sigma_T| \leq r$, this can be done in $O(r \log r)$ time and $O(r)$ space using any standard sorting algorithm.

Computing $\mathcal{M}^{(2)}$

To compute $\mathcal{M}^{(2)}$, we check for each $0 \leq i \leq r - 2$, whether there exists a MUS that starts at $epos_T(i)$ and insert it in $\mathcal{M}^{(2)}$ if there is. More specifically, we first compute y such that $T[epos_T(i)..y]$ is right minimal unique. Next, we check whether $T[epos_T(i) + 1..y]$ is unique or not, and if not, we have that $[epos_T(i), y]$ is also left minimal unique and thus is a MUS.

Let $q = \mathsf{tRLESA}_T^{-1}[i]$. By Observation 4.1, we have that $l = \max\{\mathsf{tRLELCP}_T[q], \mathsf{tRLELCP}_T[q + 1]\}$ is the length of the longest repeat of T that starts at $epos_T(i)$. This implies that $T[epos_T(i)..epos_T(i) + l]$ is right minimal unique. Thus, given the $\mathsf{tRLELCP}_T$ array, $y = epos_T(i) + l$ can be computed in constant time. Next, to determine whether $T[epos_T(i) + 1..y]$ is unique or not, we compute y' such that $T[epos_T(i) + 1..y']$ is right minimal unique. Then, $[epos_T(i) + 1, y]$ is unique iff $y' \leq y$. Noticing that $epos_T(i) + 1 = bpos_T(i + 1)$, we can compute y' as follows. Let $q = \mathsf{tRLESA}_T^{-1}[i + 1]$ and $x = \mathsf{EXP}_T[q]$. We compute $l' = x - 1 + trle\_lce_T(q, q')$, where $q' = exp\_pos(q, x)$. By definition, we have that l' is the length of the longest repeat of T that starts at $bpos_T(i + 1)$. Thus, $y' = bpos_T(i + 1) + l'$. By Lemma 4.2, this can be computed in $O(r \log r)$ total time and $O(r)$ space for all i.

Computing $\mathcal{M}^{(3)}$

For each $0 \leq i < r - 1$, we will compute the elements of $\mathcal{M}^{(3)}$ that start in the i-th run. Let $s = bpos_T(i)$ and we repeat the following while $s < epos_T(i)$. First, compute y such that $T[s..y]$ is right minimal unique. If such y does not exists, i.e., $T[s..]$ is not unique, then we are done. If y does exist, $y \geq epos_T(i)$ since, as noted earlier, no proper substring of a run can be unique. If $y = epos_T(i)$, we must have that $s = bpos_T(i)$ and $[s, y]$ is a MUS in $\mathcal{M}^{(1)}$ and not

in $\mathcal{M}^{(1)}$; thus we simply increment s by 1 and repeat the process. Otherwise, if $y > epos_T(i)$, we try to find x such that $T[x..y]$ is left minimal unique. Then, by definition, $[x, y]$ is a MUS. If $x < epos_T(i)$, then we have that $[x, y]$ is a MUS in $\mathcal{M}^{(3)}$, and since there can be no other MUS that starts in the interval $[s, x]$, we set $s = x + 1$ and repeat the process. Otherwise, if $x \geq epos_T(i)$, then $[x, y]$ is either a MUS in $\mathcal{M}^{(2)}$ or does not start in the i-th run, so we are finished for the current value of i. Because we obtain one distinct MUS each time we determine y and x, the above process is repeated for a total of $O(r)$ times for all i by Lemma 4.5. What remains is how to determine y and x.

Whether $y = epos_T(i)$ or not can be determined by checking if $[s, epos_T(i)]$ is a MUS in $\mathcal{M}^{(1)}$ as described in Section 4.2.2. Next, we assume $y \geq epos_T(i) + 1 = bpos_T(i + 1)$. Let $q = \mathsf{tRLESA}_T^{-1}[i]$, $q' = exp\_pos(q, epos_T(i) - s + 1)$. If q' is nil, this implies that no run other than the i-th one contains a run of character $T[epos_T(i)]$ with length at least $epos_T(i) - s + 1$. Since $y > epos_T(i)$, we have that $T[s..epos_T(i)]$ is not unique but $T[s..bpos_T(i + 1)]$ is unique and thus, $y = bpos_T(i+1)$. Otherwise, if q' is not nil, then, we have that $epos_T(i) - s + l$, where $l = trle\_lce_T(q, q')$ is the length of the longest repeat of T that starts at s. Therefore, we have $y = epos_T(i) + l$. From the above arguments and Lemma 4.2, y can be determined in $O(\log r)$ time. Whether $x \geq epos_T(i)$ or not can be determined by the arguments for checking whether $[epos_T(i), y]$ is a MUS in $\mathcal{M}^{(2)}$, as described in Section 4.2.2. Next, we assume $x < epos_T(i)$. Then, $T[epos_T(i)..y]$ is a repeat. Let $q = \mathsf{tRLESA}_T^{-1}(i)$, $q' = lce\_pos(q, y - epos_T(i) + 1)$. From the definition of lce_pos, we have $x = epos_T(i) - \mathsf{EXP}_T[q'] + 1$. Thus, from the above arguments and Lemma 4.3, x can be determined in $O(\log r)$ time.

The arguments from Sections 4.2.2-4.2.2 lead to the following lemma.

Lemma 4.7. *For any string T, the set MUS_T can be computed from $RLE(T)$ in $O(r \log r)$ time using $O(r)$ space, where $r = |RLE(T)|$.*

4.3 Solution to the SUS Problem

4.3.1 Data Structure

Our data structure consists of three arrays: X_T, Y_T, and MUSlen_T. Arrays X_T and Y_T are arrays of size $|\mathsf{MUS}_T|$ such that for any $0 \leq i \leq |\mathsf{MUS}_T| - 1$, $[\mathsf{X}_T[i], \mathsf{Y}_T[i]]$ is the i-th MUS in order of their start position in T. Also, let the array $\mathsf{MUSlen}_T[i] = \mathsf{Y}_T[i] - \mathsf{X}_T[i] + 1$ hold the length of each MUS. Arrays X_T and Y_T are preprocessed for Succ and Pred queries, and MUSlen_T

is preprocessed for RmQ queries. From arguments in previous sections, the preprocessing can clearly be done in a total of $O(r \log r)$ time and $O(r)$ space.

4.3.2 Answering Queries

For any two intervals $[s, t]$ and $[x, y]$, let $cover([s, t], [x, y])$ be the smallest interval that contains both $[s, t], [x, y]$, i.e., $cover([s, t], [x, y]) = [\min\{s, x\}, \max\{t, y\}]$.

Given a query interval $[s, t]$, let $i = \mathsf{Pred}_{\mathsf{Y}_T}(t)$ and $j = \mathsf{Succ}_{\mathsf{X}_T}(s)$. Clearly, all SUSs that contain interval $[s, t]$ are contained in the set $\{|cover([s, t], [\mathsf{X}_T[r], \mathsf{Y}_T[r]])| \mid i \leq r \leq j\}$. Thus, it suffices to find the intervals of smallest size in this set, i.e., if $p \in \arg\min\{|cover([s, t], [\mathsf{X}[r], \mathsf{Y}[r]])| \mid i \leq r \leq j\}$, then $cover([s, t], [\mathsf{X}_T[p], \mathsf{Y}_T[p]])$ is a SUS. Notice that for all $i < r < j$, we have that $cover([s, t], [\mathsf{X}_T[r], \mathsf{Y}_T[r]]) = [\mathsf{X}_T[r], \mathsf{Y}_T[r]]$. Thus, the shortest of these can be found by considering $cover([s, t], [\mathsf{X}_T[i], \mathsf{Y}_T[i]])$, $cover([s, t], [\mathsf{X}_T[j], \mathsf{Y}_T[j]])$, and performing an RmQ query on MUSlen_T. An example is shown in Fig. 4.2. For finding a single SUS, the query time is dominated by the Pred and Succ queries. To output all SUSs that contain $[s, t]$, recursive RmQ on sub-intervals of MUSlen_T can be conducted in constant time per output, in order to find all the shortest intervals in the range $[i, j]$. Thus, the total query time is the time for a single predecessor query plus $O(k)$, where k is the total number of SUSs that are output.

Let $\pi_s(n, r)$, $\pi_c(n, r)$, $\pi_q(n, r)$ are, respectively, the size, construction time, and query time for a predecessor/successor query data structure of r elements for the universe of $[0, n - 1]$. Putting everything together, we have proved the following theorem:

Theorem 4.1. *Given $RLE(T)$ of size r representing a string T of length n, we can compute in $O(r \log r + \pi_c(n, r))$ time a data structure of size $O(r + \pi_s(n, r))$ which answers SUS queries for any interval $[s, t] \subseteq [0, n - 1]$ in $O(\pi_q(n, r) + k)$ time, where k is the number of SUSs to output.*

Using known results for predecessor/successor queries [6], we obtain the following corollary.

Corollary 4.1. *Given $RLE(T)$ of size r representing a string T of length n, we can compute in $O(r \log r)$ time a data structure of size $O(r)$ which answers SUSs queries for any interval $[s, t] \subseteq [0, n - 1]$ in $O(\sqrt{\log r / \log\log r} + k)$ time, where k is the number of such SUSs.*

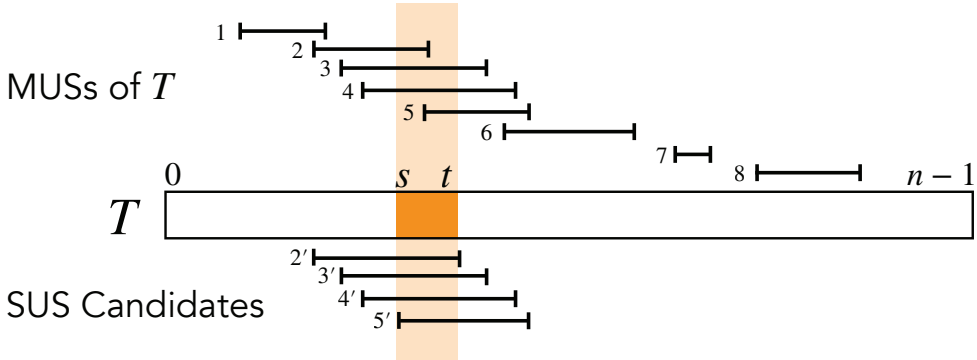

Figure 4.2: Finding SUSs that contains query interval $[s, t]$. The SUS must be either a MUS that completely contains $[s, t]$ (MUS 3,4), or, it must be an interval that covers both $[s, t]$ and the preceding MUS (MUS 2) or succeeding MUS (MUS 5). Of these, the intervals with shortest length are the SUSs that contain $[s, t]$.

4.4 Conclusions and Open Question

We considered the problem of finding all shortest unique substrings (SUSs) of a string T given as the run-length encoding (RLE) of size r. We showed that we can preprocess the RLE in $O(r \log r)$ time and $O(r)$ space so that subsequent SUS queries for T can be answered in $O(\sqrt{\log r / \log\log r} + k)$ time, where k is the number of outputs for the query interval. Notice that none of the preprocessing time, space requirement, or query time depends on the original length n of the string T. This efficiency was achieved by a non-trivial use of the suffix arrays for RLE strings and by revealing combinatorial properties of MUSs and SUSs on RLE strings.

The $\sqrt{\log r / \log\log r}$ term in our query time is due to the use of the $O(r)$-space dynamic predecessor/successor data structure by Beame and Fich [6]. They also showed that for a static set $\mathcal{A}$ of r integers from the universe $[0, n-1]$, any predecessor/successor data structure for $\mathcal{A}$ of polynomial size in r must use $\Omega(\sqrt{\log r / \log\log r})$ query time (Corollary 3.10 of [6]). Notice that once we build arrays X_T and Y_T, they will remain static. Hence, we cannot hope for faster SUS query time as long as we use predecessor/successor queries to find a MUS for a given interval. Thus, an interesting open question is whether there exists a data structure of size $O(r)$ that can efficiently answer SUS queries without using predecessor/successor queries.

Chapter 5

Space-Efficient Algorithms for Computing Minimal/Shortest Unique Substrings

In this chapter, we propose the following two data structures for SUS problems:

(A) A data structure of size $2n + 2m + o(n)$ bits answering an interval SUS query in $O(k)$ time, where m is the number of *minimal unique substrings* of the input string, and k is the number of SUSs of T for the respective query interval (Theorem 5.1).

(B) A data structure of size $\lceil(\log_2 3 + 1)n\rceil + o(n)$ bits answering a point SUS query in $O(k)$ time, where k is the number of SUSs of T for the respective query point (Theorem 5.2).

Instead of outputting the answer as a list of substrings of T, it is sometimes sufficient to output only the intervals corresponding to the respective substrings. In such a case, both data structures can answer a query *without* the need of the input string. The data structure (A) is the first data structure of size $O(n)$ bits for the interval SUS problem. Also, the data structure (B) is the first data structure of size $O(n)$ bits for the point SUS problem, returning *all* SUSs for a given query position. Notice that the data structure of Ganguly et al. [21] uses $4n + o(n)$ bits of space, but returns only one SUS for a point SUS query.

5.1 Computing MUSs in Compact Space

For computing SUSs efficiently, it is advantageous to have a data structure available that can retrieve MUSs starting or ending at specific positions, as the following lemma gives a crucial connection between MUSs and SUSs:

Lemma 5.1 ([55, Lemma 2])**.** *Every point SUS contains exactly one MUS.*

Fig. 5.1 gives an overview of our introduced data structure and shows the connections between this section and the following sections that focus on our two SUS problems. For our data structure retrieving MUSs, we propose a compact representation and an algorithm to compute this representation space-efficiently. Our data structure is based on the following two bit arrays MB_T and ME_T of length n with the properties that

- $\mathsf{MB}_T[i] = 1$ iff i is the beginning position of a MUS, and
- $\mathsf{ME}_T[i] = 1$ iff i is the ending position of a MUS.

For the rest of this chapter, let m be the number of MUSs in T. We rank the MUSs by their starting positions in the text, such that the j-th MUS starts before the $(j+1)$-th MUS, for every integer j with $0 \leq j \leq m-2$.

Since MUSs are not nested (see Lemma 2.1), the number of 1's in MB_T and ME_T is exactly m. Hence, the starting position, the ending position, and the length of the j-th MUS can be computed with rank/select queries for every integer j with $0 \leq j \leq m-1$. How MB_T and ME_T can be computed is shown in the following lemma:

Lemma 5.2. *Let $\mathcal{D}_T$ be a data structure that can access $\mathsf{ISA}_T[i]$ and $\mathsf{LCP}_T[i]$ in $\pi_a(n)$ time for every position i with $0 \leq i \leq n-1$. Suppose that we can construct it in $\pi_c(n)$ time with $\pi_s(n)$ bits of working space including the space for $\mathcal{D}_T$. Then MB_T and ME_T can be computed in $O(\pi_c(n) + n \cdot \pi_a(n))$ total time while using $2n + \pi_s(n)$ bits of total working space including the space for MB_T and ME_T.*

Proof. Given a text position i with $0 \leq i \leq n-1$, $T[i..i+\ell_i-1]$ with $\ell_i = \max\{\mathsf{LCP}_T[\mathsf{ISA}_T[i]], \mathsf{LCP}_T[\mathsf{ISA}_T[i]+1]\}$ is the longest repeating substring starting at i. If we extend this substring by the character to its right, it becomes unique. Thus, $T[i..i+\ell_i]$ is the shortest unique substring starting at i, except for the case that $i+\ell_i-1 = n-1$ as we cannot extend it to the right (hence, there is no unique substring starting at i in this case). Additionally, the substring $T[i..i+\ell_i]$ is a MUS iff $T[i+1..i+\ell_i]$ is not unique (we already checked that $T[i..i+\ell_i-1]$ is not unique). $T[i+1..i+\ell_i]$ is not unique iff $\ell_i \leq \ell_{i+1}$ since $T[i+1..i+1+\ell_{i+1}]$ is the shortest unique substring starting at $i+1$. Since each ℓ_i can be computed in $O(\pi_a(n))$ time for every $0 \leq i \leq n-1$, the starting and ending positions of all MUSs (and hence, MB_T and ME_T) can be computed in $O(n \cdot \pi_a(n))$ time by a linear scan of the text. Therefore, the total computing time is $O(\pi_c(n) + n \cdot \pi_a(n))$ and the total working space is $2n + \pi_s(n)$ bits including the space for MB_T and ME_T. □

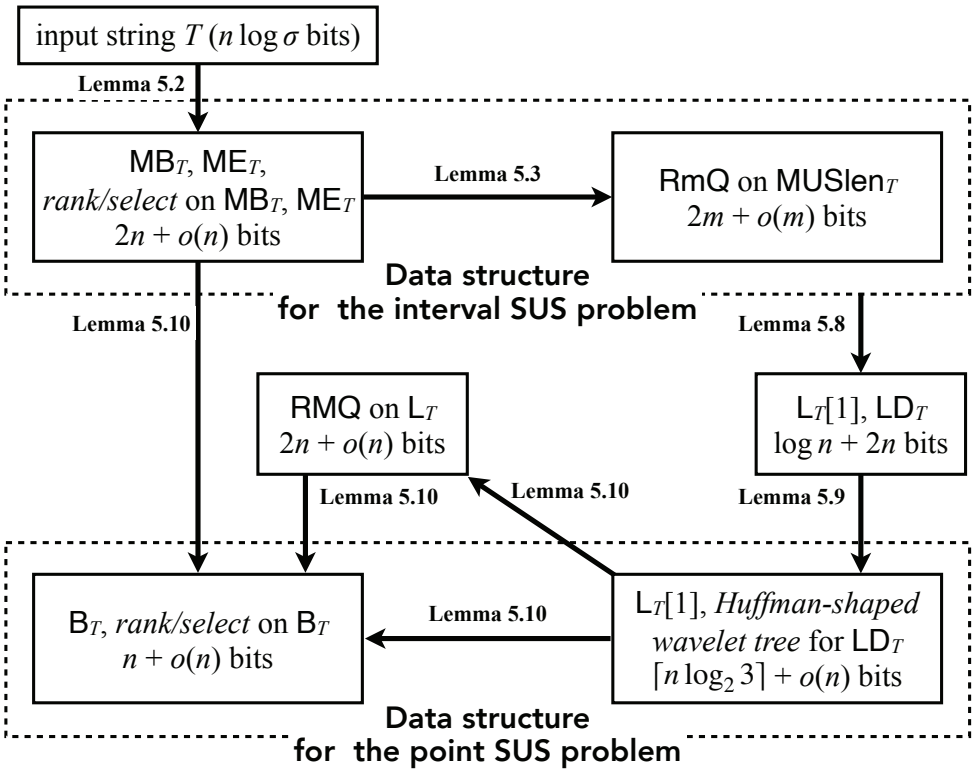

Figure 5.1: Overview of the data structures proposed for solving the interval SUS and point SUS problem. Nodes are data structures. Edges of the same label (labeled by a certain lemma) describe an algorithm taking a set of input data structures to produce a data structure.

5.2 Compact Data Structure for the Interval SUS Problem

In this section, we propose a compact data structure for the interval SUS problem. It is based on the data structure of Chapter 4, which we review in the following. We subsequently provide a compact representation of this data structure.

Data Structures. The data structure described in Chapter 4 consists of three arrays, each of length m: X_T, Y_T, and MUSlen_T. The arrays X_T and Y_T store, respectively, the beginning positions and ending positions of all MUSs sorted by their beginning positions such that the interval $[\mathsf{X}_T[i], \mathsf{Y}_T[i]]$ is the i-th MUS, for every integer i with $0 \leq i \leq m-1$. Further, $\mathsf{MUSlen}_T[i] = \mathsf{Y}_T[i] - \mathsf{X}_T[i] + 1$ stores the length of i-th MUS. During a preprocessing phase, X_T and Y_T are endowed with a successor and a predecessor data structure, respectively. Further, MUSlen_T is endowed with an RmQ data structure.

Answering Queries. Given a query interval $[s, t]$, let $L = \mathsf{Pred}_{Y_T}(t)$ be the index in Y_T of the largest ending position of a MUS that is at most t, and $R = \mathsf{Succ}_{X_T}(s)$ be the index in X_T of the smallest starting position of a MUS that is at least s. Then, $\mathsf{SUS}_T([s, t]) \subset \{cover([s, t], [X_T[i], Y_T[i]]) \mid L \leq i \leq R\}$. That is because the shortest intervals in $\{cover([s, t], [X_T[i], Y_T[i]]) \mid L \leq i \leq R\}$ correspond to the shortest unique substrings (SUSs) among all substrings covering the interval $[s, t]$. Thus, one of the SUSs for $[s, t]$ can be detected by considering $cover([s, t], [X_T[L], Y_T[L]])$ (as a candidate for the leftmost SUS), $cover([s, t], [X_T[R], Y_T[R]])$ (as a candidate for the rightmost SUS), and $\mathsf{RmQ}_{\mathsf{MUSlen}_T}(L+1, R-1)$. To output all SUSs, it is sufficient to answer RmQ queries on subintervals of $\mathsf{MUSlen}_T[L + 1..R - 1]$ recursively. In detail, suppose that there is a MUS in $\mathsf{MUSlen}_T[L + 1..R - 1]$ that is a SUS for $[s, t]$. Further suppose that this is the j-th MUS having length k. Then we query $\mathsf{MUSlen}_T[L + 1..j - 1]$ and $\mathsf{MUSlen}_T[j + 1..R - 1]$ for all other MUSs of minimal length k.

Compact Representation. Having the two bit arrays MB_T and ME_T of Section 5.1, we can simulate the three arrays X_T, Y_T, and MUSlen_T. By endowing these two bit arrays with rank/select data structures of Lemma 2.3, we can compute rank/select in constant time, which allows us to compute the value of $\mathsf{X}_T[p]$, $\mathsf{Y}_T[p]$, $\mathsf{MUSlen}_T[p]$, $\mathsf{Pred}_{Y_T}(q)$ and $\mathsf{Succ}_{X_T}(q)$ for every index p with $0 \leq p \leq m-1$ and every text position q with $0 \leq q \leq n-1$ in constant time while using only $2n + o(n)$ bits of total space. By endowing MUSlen_T with the RmQ data structure of Lemma 2.2, we can answer an RmQ query on MUSlen_T in constant time. This data structure takes $2m + o(m)$ bits of space. Altogether, with these data structures we yield the following theorem:

Theorem 5.1. *For the interval SUS problem, there exists a data structure of size $2n+2m+o(n)$ bits that can answer an interval SUS query in $O(k)$ time, where k is the number of SUSs of T for the respective query interval.*

Also, the data structure can be constructed space-efficiently:

Lemma 5.3. *Given MB_T and ME_T, the data structure proposed in Theorem 5.1 can be constructed in $O(n)$ time using $2m + o(n)$ bits of total working space, which includes the space for this data structure.*

Proof. The data structure proposed in Theorem 5.1 consists of the two bit arrays MB_T, ME_T, and an RmQ data structure on MUSlen_T, which is simulated by rank/select data structures on MB_T and ME_T. Since MB_T and ME_T are already given, it is left to endow MB_T and ME_T

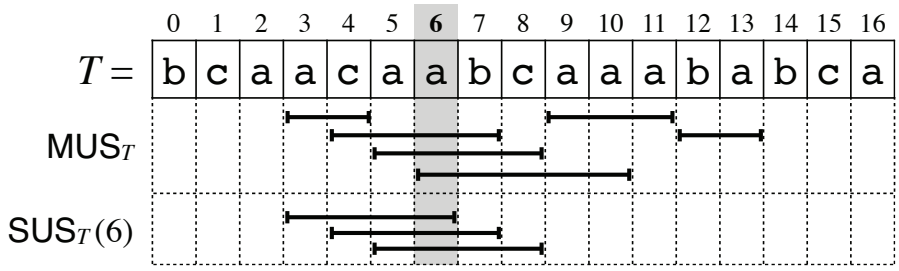

Figure 5.2: The string $T = \texttt{bcaacaabcaaababca}$ and the sets MUS_T and $\mathsf{SUS}_T(6)$. The substrings $T[3..6] = \texttt{acaa}$, $T[4..7] = \texttt{caab}$, and $T[5..8] = \texttt{aabc}$ are SUSs for the query position 6. The leftmost/rightmost SUS and MUS for $p = 6$ are $\text{lmSUS}_T(p) = [3, 6]$, $\text{lmMUS}_T(p) = [3, 4]$, $\text{rmSUS}_T(p) = [5, 8]$, and $\text{rmMUS}_T(p) = [5, 8]$.

with rank/select data structures (using Lemma 2.3), and to compute the RmQ data structure on MUSlen_T (using Lemma 2.2). □

5.3 Compact Data Structure for the Point SUS Problem

Before solving the point SUS problem, we borrow some additional notations from Tsuruta et al. [55] to deal with point SUS queries. This is necessary since some of the MUSs never take part in finding a SUS such that there is no meaning to compute and store them. Since we want to provide an output-sensitive algorithm answering a query in optimal time, we only want to store MUSs that are candidates for being a SUS.

We say that the interval $[x, y] \in \mathsf{MUS}_T$ is a *meaningful* MUS if $T[x..y]$ is a substring of (or equal to) a point SUS, i.e., $cover([x, y], p) \in \mathsf{SUS}_T(p)$ for a position p. Also, we say that the interval $[x, y] \in \mathsf{MUS}_T$ is a *meaningless* MUS if $[x, y]$ is not a meaningful MUS.

Let $\text{lmSUS}_T(p)$ denote the interval in $\mathsf{SUS}_T(p)$ with the leftmost starting position, and let $\text{lmMUS}_T(p)$ denote the MUS contained in $\text{lmSUS}_T(p)$. We say that $\text{lmSUS}_T(p)$ is the *leftmost SUS* for p, and $\text{lmMUS}_T(p)$ is the *leftmost MUS* for p. Similarly, we define the *rightmost SUS* $\text{rmSUS}_T(p)$ and the *rightmost MUS* $\text{rmMUS}_T(p)$ for p by symmetry. See Fig. 5.2 for an example for the leftmost/rightmost SUS and MUS. Let L_T be an array of length n such that $\mathsf{L}_T[i]$ is the length of a SUS[1] of T containing i for each position i with $0 \leq i \leq n-1$. Let B_T be a bit array of length n such that $\mathsf{B}_T[i] = 1$ iff i is the beginning position of a meaningful MUS of T.

From the definition of L_T, we yield the following observation:

[1] Although there can be multiple SUSs containing i, their lengths are all equal.

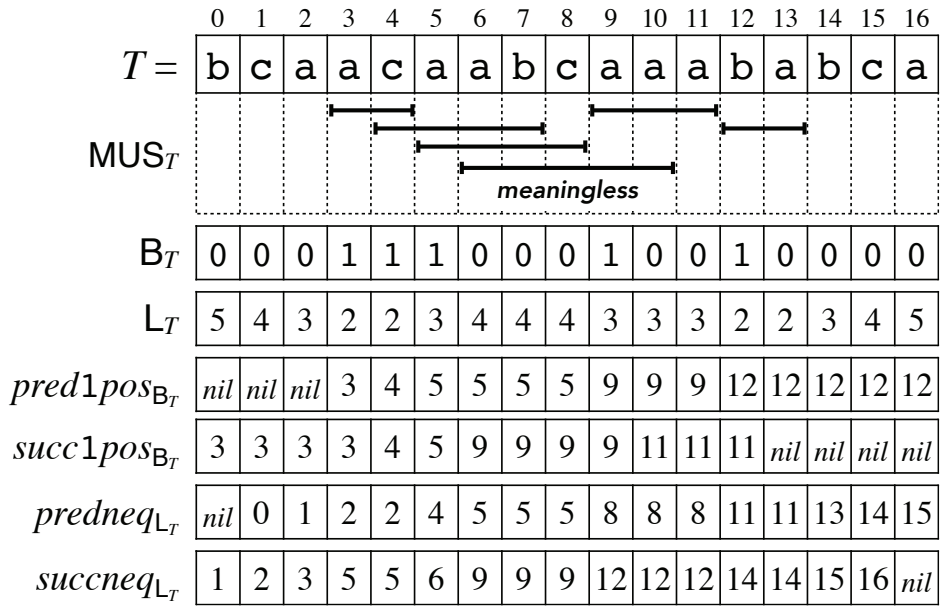

Figure 5.3: $\mathsf{MUS}_T, \mathsf{B}_T$, L_T, and the four functions defined in at the beginning of Section 5.3 for the string $T = \texttt{bcaacaabcaaababca}$. $\mathsf{B}_T[6] = 0$ because the MUS $T[6..10] = \texttt{abcaa}$ is meaningless.

Observation 5.1. *For every position p with $0 \leq p \leq n-1$ and every interval $[x, y] \in \mathsf{SUS}_T(p)$, $p - \mathsf{L}_T[p] + 1 \leq x \leq p \leq y \leq p + \mathsf{L}_T[p] - 1$.*

Next, we define the following four functions related to L_T and B_T. For a position q with $0 \leq q \leq n-1$ let

- $pred1pos_{\mathsf{B}_T}(q) = \max\{i \mid i \leq q \text{ and } \mathsf{B}_T[i] = 1\}$,
- $succ1pos_{\mathsf{B}_T}(q) = \min\{i \mid i \geq q \text{ and } \mathsf{B}_T[i] = 1\}$,
- $predneq_{\mathsf{L}_T}(q) = \max\{i \mid i < q \text{ and } \mathsf{L}_T[i] \neq \mathsf{L}_T[q]\}$, and
- $succneq_{\mathsf{L}_T}(q) = \min\{i \mid i > q \text{ and } \mathsf{L}_T[i] \neq \mathsf{L}_T[q]\}$.

For all four functions, we stipulate that $\min\{\} = \max\{\} = nil$. See Fig. 5.3 for an example of the arrays and functions defined above.

5.3.1 Finding SUSs with L and B

Our idea is to answer point SUS queries with L_T and B_T. For that, we first think about how to find the leftmost and rightmost SUS for a given query (Observation 5.1 gives us the range in

which to search). Having this leftmost and the rightmost SUS, we can find all other SUSs with B_T marking the beginning positions of the meaningful MUSs that correspond to the SUSs we want to output. Before that, we need some properties of L_T that help us to prove the following lemmas in this section: Lemma 5.4 gives us a hint on the shape of L_T, while Lemma 5.5 shows us how to find SUSs based on two consecutive values of L_T with a connection to MUSs.

Lemma 5.4. $|\mathsf{L}_T[p] - \mathsf{L}_T[p+1]| \leq 1$ *for every position* p *with* $0 \leq p \leq n-2$.

Proof. Let $\ell = \mathsf{L}_T[p]$ and $\ell' = \mathsf{L}_T[p+1]$. From the definition of L_T, there exists a unique substring of length ℓ containing the position p. If $\ell < \ell'$, there is no unique substring of length ℓ containing $p+1$. Thus, $T[p-\ell+1..p]$ is unique, and consequently $T[p-\ell+1..p+1]$ is also unique. Hence, $\ell' = \ell + 1$. Similarly, in the case of $\ell > \ell'$, it can be proven that $\ell' = \ell - 1$. □

Lemma 5.5. *Let* p *be a position with* $0 \leq p \leq n-2$*, and let* $\ell = \mathsf{L}_T[p]$.
If $\mathsf{L}_T[p+1] = \ell + 1$*, then*

- $T[p-\ell+1..p] \in \mathsf{SUS}_T(p)$,
- $T[p-\ell+1..p+1] \in \mathsf{SUS}_T(p+1)$*, and*
- $p-\ell+1$ *is the starting position of a MUS of* T.

If $\mathsf{L}[p+1] = \ell - 1$ *then*

- $T[p..p+\ell-1] \in \mathsf{SUS}_T(p)$,
- $T[p+1..p+\ell-1] \in \mathsf{SUS}_T(p+1)$*, and*
- $p+\ell-1$ *is the ending position of a MUS of* T.

Proof. First, we consider the case that $\mathsf{L}_T[p+1] = \ell + 1$. From the proof of Lemma 5.4, $T[p-\ell+1..p]$ and $T[p-\ell+1..p+1]$ are unique substrings in T. Thus, $T[p-\ell+1..p] \in \mathsf{SUS}_T(p)$ and $T[p-\ell+1..p+1] \in \mathsf{SUS}_T(p+1)$. Since every point SUS contains exactly one MUS (cf. Lemma 5.1), there exists a MUS $[b, e] \subset [p-\ell+1, p]$. Assume that $b > p-\ell+1$, then $T[b..p]$ is the shortest unique substring among all substrings containing the text position p. Its length is $p-b+1 < \ell$. This contradicts that $T[p-\ell+1..p] \in SUS_T(p)$, and therefore $b = p-\ell+1$ must hold. The remaining case $\mathsf{L}_T[p+1] = \ell - 1$ can be proven analogously by symmetry. □

In the following two lemmas (Lemmas 5.6 and 5.7), we focus on finding the leftmost SUS and the rightmost SUS for a given query point. That is because the leftmost SUS and the

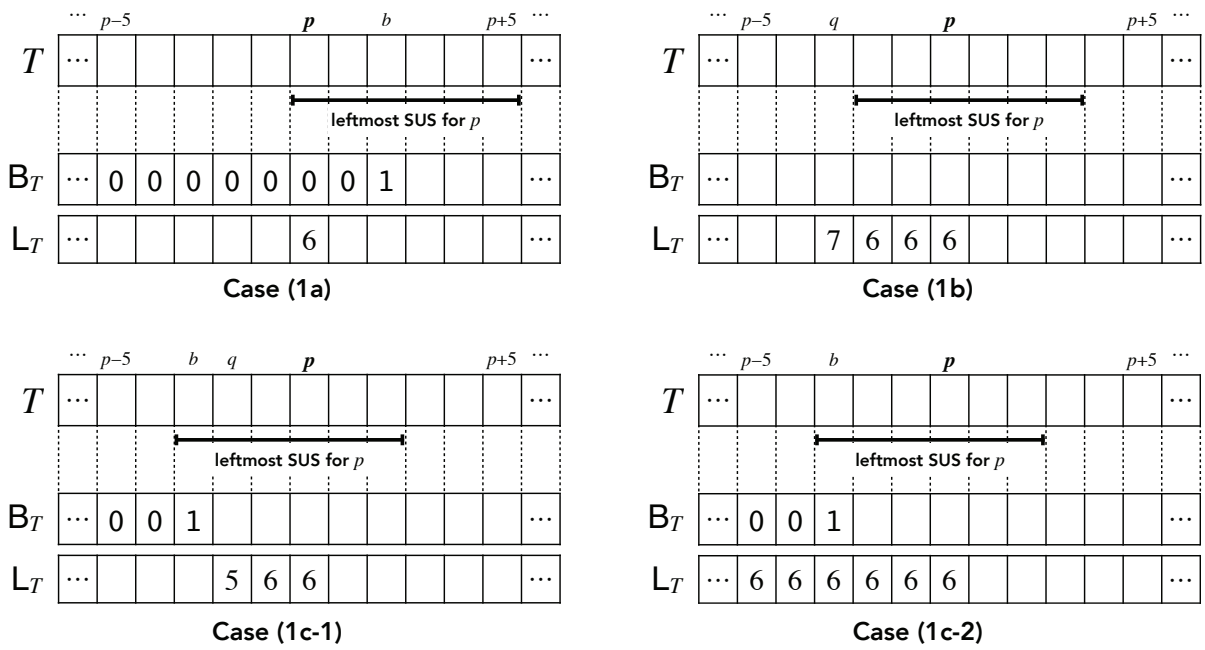

Figure 5.4: Example of the proof of Lemma 5.6 with $\mathsf{L}_T[p] = \ell = 6$. The example (as well as all later examples in this section) still works when replacing the number $\ell = 6$ with another number as long as the relative differences to the other entries in L_T and the search range in T is kept.

rightmost SUS give us an interval containing the starting positions of the remaining SUSs we want to report[2].

Lemma 5.6. *Let p be a position with $0 \leq p \leq n-1$, and let $\ell = \mathsf{L}_T[p]$, $b = succ1pos_{\mathsf{B}_T}(\max\{1, p-\ell+1\})$, and $q = predneq_{\mathsf{L}_T}(p)$. Then, $b \leq \min\{p+\ell-1, n\}$ and*

$$\mathrm{lmSUS}_T(p) = \begin{cases} [p, p+\ell-1] & \textit{if } b \geq p, & (5.1a) \\ [q+1, q+\ell] & \textit{if } b < p,\ q \geq p-\ell+1, \textit{ and } \mathsf{L}_T[q] > \ell, & (5.1b) \\ [b, b+\ell-1] & \textit{otherwise.} & (5.1c) \end{cases}$$

Proof. If $\ell = 1$, it is clear that the interval $[p, p]$ of length 1 is a MUS of T, thus $b = p$ and $\mathrm{lmSUS}_T(p) = [p, p]$. For the rest of the proof, we focus on the case that $\ell \geq 2$. Since $\mathsf{L}_T[p] = \ell$, there exists a unique substring of length ℓ containing the position p, and there exists at least one MUS that is a subinterval of $[p-\ell+1, p+\ell-1]$. Thus, $b \leq \min\{p+\ell-1, n-1\}$. See Fig. 5.4 for an illustration of each of the above cases we consider in the following:

(1a) Assume that there exists a unique substring $T[p'..p'+\ell-1]$ containing the position p with

[2]The actual reporting of those SUSs is done in Lemma 5.11

$p' < p$. Since $b \geq p > p'$, $T[p'+1..p'+\ell-1]$ is also unique and contains position p. It contradicts $\mathsf{L}_T[p] = \ell$; therefore, $\text{lmSUS}_T(p) = [p, p+\ell-1]$.

(1b) From the definition of q and Lemma 5.4, $\mathsf{L}_T[q] = \ell+1$ and $\mathsf{L}_T[q+1] = \ell$. From Lemma 5.5, $[q+1, q+\ell]$ is unique. Also, $[q+1, q+\ell] \in \mathsf{SUS}_T(p)$ because $p \in [q+1, q+\ell]$. Since $\mathsf{L}_T[q] = \ell+1$, there is no unique substring that contains the position q and is shorter than $\ell+1$. Therefore, $\text{lmSUS}_T(p) = [q+1, q+\ell]$.

(1c) We divide this case into two subcases:

(1c-1) $b < p$ and $q \geq p-\ell+1$ and $\mathsf{L}_T[q] < \ell$, or

(1c-2) $b < p$ and $q < p-\ell+1$.

In Subcase (1c-1), from the definition of q and Lemma 5.4, $\mathsf{L}_T[q] = \ell-1$ and $\mathsf{L}_T[i] = \ell$ for all $i \in [q+1, p]$. From Lemma 5.5, the interval $[q-\ell+2, q]$ of length $\ell-1$ is unique. Since $[p-\ell+1, q] \subset [q-\ell+2, q]$, $\mathsf{L}_T[i] \leq \ell-1$ for all $i \in [p-\ell+1, q]$. In Subcase (1c-2), it is clear that $\mathsf{L}_T[i] = \ell$ for all $i \in [p-\ell+1, p]$. Therefore, $\mathsf{L}_T[i] \leq \ell$ for all $i \in [p-\ell+1, p]$ in both subcases.

Let e be the ending position of the meaningful MUS $[b, e]$ starting at the position b, and $\ell' = e-b+1$ be the length of this MUS. We assume $\ell' > \ell$ for the sake of contradiction (and thus $[b, e]$ cannot be $\text{lmMUS}_T(p)$ whose length is at most ℓ). Since $b \geq p-\ell+1$ and $\ell' > \ell$, $e > p$ must hold. Let $[b', e'] = \text{lmMUS}_T(p)$. Since (a) there is no interval $[x, y] \in \mathsf{SUS}_T(p)$ such that $x < \min\{b, p\}$, and (b) MUSs cannot be nested, it follows that $b' > b$ and $e' > e$. Thus, $\text{lmSUS}_T(p) = \mathit{cover}([b', e'], p) = [e'-\ell+1, e']$ and $\mathsf{L}_T[i] \leq \ell$ for all $p \leq i \leq e'$. Since $[b, e] \subset [p-\ell+1, e']$, $\mathsf{L}_T[i] \leq \ell$ for all $b \leq i \leq e$. This contradicts that the MUS $[b, e]$ of length $\ell' > \ell$ is a meaningful MUS. Therefore, $\ell' \leq \ell$ and $\text{lmSUS}_T(p) = \mathit{cover}([b, e], p) = [b, b+\ell-1]$.

□

From Lemma 5.6 we yield the following corollary:

Corollary 5.1. *If we can compute* $\mathsf{L}_T[i]$, $\mathit{predneq}_{\mathsf{L}_T}(i)$ *and* $\mathit{succ1pos}_{\mathsf{B}_T}(i)$ *in constant time for each* i *with* $0 \leq i \leq n-1$, *we can compute* $\text{lmSUS}_T(p)$ *in constant time for each position* p *with* $0 \leq p \leq n-1$.

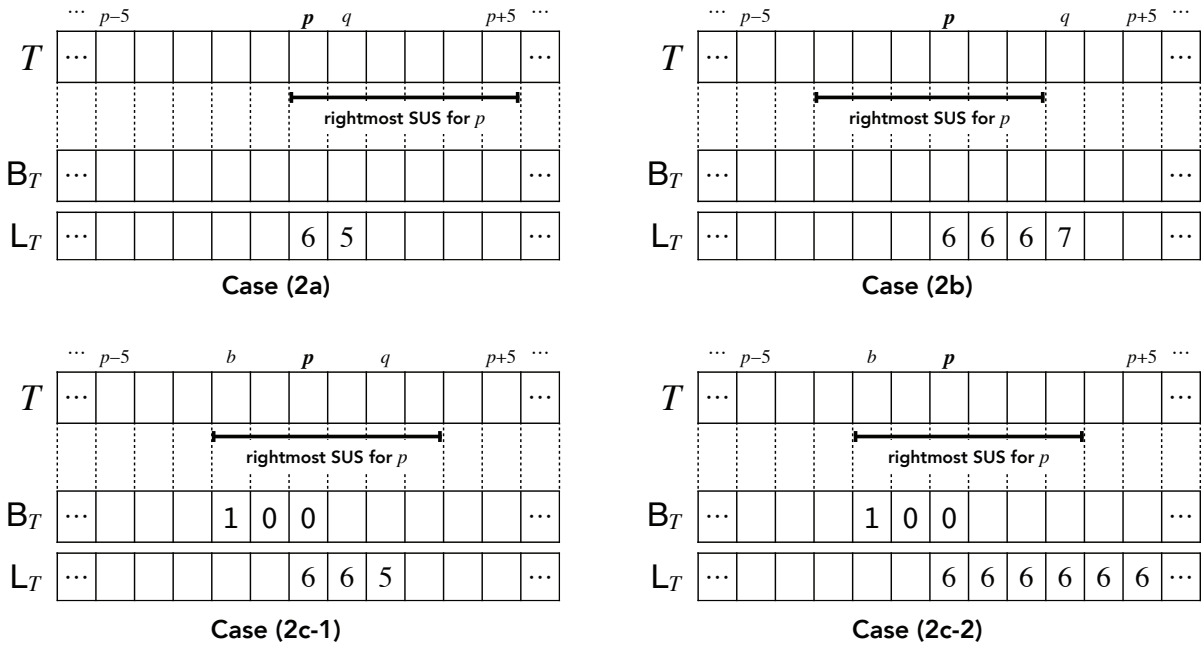

Figure 5.5: Example of the proof of Lemma 5.7 with $\mathsf{L}_T[p] = \ell = 6$.

Lemma 5.7. *Let p be a position with $0 \leq p \leq n-1$, and let $\ell = \mathsf{L}_T[p]$, $q = succneq_{\mathsf{L}_T}(p)$, and $b = pred1pos_{\mathsf{B}_T}(p)$. Then,*

$$\mathrm{rmSUS}_T(p) = \begin{cases} [p, p+\ell-1] & \textit{if } q = p+1 \textit{ and } \mathsf{L}_T[q] < \ell, & (5.2a) \\ [q-\ell, q-1] & \textit{if } q \leq p+\ell-1 \textit{ and } \mathsf{L}_T[q] > \ell, & (5.2b) \\ [b, b+\ell-1] & \textit{otherwise.} & (5.2c) \end{cases}$$

Proof. If $\ell = 1$, it is clear that the interval $[p, p]$ of length 1 is a MUS of T, thus $b = p$ and $\mathrm{rmSUS}_T(p) = [p, p]$. We consider the condition of $\ell \geq 2$. See Fig. 5.5 for an illustration of each of the above cases we consider in the following:

(2a) From Lemma 5.5, $[p, p+\ell-1]$ is a SUS for p, which is by definition the rightmost one.

(2b) In this case, $\mathsf{L}_T[q] = \ell + 1$ and $\mathsf{L}_T[q-1] = \ell$. From Lemma 5.5, $[q-\ell, q-1]$ is unique. Since $p \in [q-\ell, q+1]$, $[q-\ell, q-1]$ is a SUS for p. Additionally, there is no unique interval $[x, y] \in \mathsf{SUS}_T(p)$ such that $y \geq q$ because $\mathsf{L}_T[q] = \ell + 1$. Thus, $\mathrm{rmSUS}_T(p) = [q-\ell, q-1]$.

(2c) We divide this case into two subcases:

(2c-1) $p+1 < q \leq p+\ell-1$ and $\mathsf{L}_T[q] < \ell$, or

(2c-2) $q > p+\ell-1$.

In Subcase (2c-1), $\mathsf{L}_T[p+1] = \ell$ and $\mathsf{L}_T[q] = \ell - 1$ and $\mathsf{L}_T[i] = \ell$ for all $p+2 \leq i \leq q-1$. From Lemma 5.5, $[q, q+\ell-2]$ (of length $\ell-1$) is unique. Since $[q, p+\ell-1] \subset [q, q+\ell-2]$, $\mathsf{L}_T[i] \leq \ell-1$ for all $q \leq i \leq p+\ell-1$. In Subcase (2c-2), from the definition of q, $\mathsf{L}_T[i] = \ell$ for all $p \leq i \leq p+\ell-1$. Therefore, $\mathsf{L}_T[p+1] = \ell$ and $\mathsf{L}_T[i] \leq \ell$ for all integers i with $p+2 \leq i \leq p+\ell-1$ in both subcases.

For the sake of contradiction, assume that there is a MUS $[b', e']$ such that $b' > p$ and $cover([b', e'], p) = [p, e'] \in \mathsf{SUS}_T(p)$. Since $\mathsf{L}_T[p] = \ell$, $[p, e']$ is a unique substring of length ℓ. Hence, $cover([b', e'], p+1) = [p+1, e']$ is a unique substring of length $\ell - 1$. It contradicts $\mathsf{L}_T[p+1] = \ell$; therefore, the beginning position of the rightmost MUS for p is at most p. Next, we show that the MUS starting at b is the rightmost meaningful MUS for p. Let e be its ending position, and $\ell' = e - b + 1$ be its length. We assume that $\ell' > \ell$ for the sake of contradiction (and thus, $[b, e]$ is not $\mathrm{rmMUS}_T(p)$ whose length is at most ℓ). Since $\mathsf{L}_T[p] = \ell$, $b \geq p - \ell + 1$ and $e > p$. Let $[b'', e''] = \mathrm{rmMUS}_T(p)$. Since MUSs cannot be nested, $b'' < b$. Since $e'' - b'' + 1 \leq \ell$, $\mathsf{L}_T[i] \leq \ell$ for all i with $b'' \leq i \leq p+\ell-1$. We consider two cases to obtain a contradiction:

- If $e \leq p+\ell-1$ then it is clear that $\mathsf{L}_T[i] \leq \ell$ for all i with $b \leq i \leq e$. This contradicts that the MUS $[b, e]$ of length ℓ' is a meaningful MUS.
- If $e > p + \ell - 1$, it is clear that $|cover([b, e], p + \ell - 1)| = |[b, e]| = \ell'$. Since $\mathsf{L}_T[p + \ell - 1] \leq \ell$, there exists a unique substring $[s, t]$ such that $s \leq p + \ell - 1 \leq t$ and $t - s + 1 \leq \ell$. Hence, $\mathsf{L}_T[i] \leq \ell$ for all i with $s \leq i \leq t$. Since $[b, e]$ is a MUS and $p \leq s$, $b < s < e < t$. Consequently, $\mathsf{L}_T[i] \leq \ell$ for all i with $b \leq i \leq e$ and this contradicts that the MUS $[b, e]$ of length ℓ' is a meaningful MUS.

Therefore, $\ell' \leq \ell$ and $\mathrm{rmSUS}_T(p) = cover([b, e], p) = [b, b + \ell - 1]$.

□

Corollary 5.2. *If we can compute* $\mathsf{L}_T[i]$, $succneq_{\mathsf{L}_T}(i)$ *and* $pred1pos_{\mathsf{B}_T}(i)$ *in constant time for each* i *with* $0 \leq i \leq n-1$, *we can compute* $\mathrm{rmSUS}_T(p)$ *in constant time for each position* p *with* $0 \leq p \leq n-1$.

5.3.2 Compact Representations of L

We now propose a succinct representation of the array L_T consisting of the integer array LD_T of length n defined as $\mathsf{LD}_T[0] = 0$ and $\mathsf{LD}_T[i] = \mathsf{L}_T[i] - \mathsf{L}_T[i-1] \in \{-1, 0, 1\}$ for every i with $1 \leq i \leq n-1$.

No.	Process	Total working space in bits (excluding MB_T and ME_T)	
1	input MB_T, ME_T	-	
2	construct RmQ on MUSlen_T	$2m + o(n)$	Lemma 5.3
3	construct LD_T, $\mathsf{L}_T[0]$	$2n + 2m + o(n)$	Lemma 5.8
4	free RmQ on MUSlen_T	$2n + o(n)$	
5	construct *Huffman-shaped Wavelet Tree* for LD_T	$2n + \lceil n \log_2 3 \rceil + o(n)$	Lemma 5.9
6	free LD_T	$\lceil n \log_2 3 \rceil + o(n)$	
7	construct RMQ on L_T	$2n + \lceil n \log_2 3 \rceil + o(n)$	Lemma 5.10
8	construct B_T	$3n + \lceil n \log_2 3 \rceil + o(n)$	Lemma 5.10

Table 5.1: Working space used during the construction of the data structure proposed in Theorem 5.2. We can free up space of no longer needed data structures between several steps. See also Fig. 5.1 for the dependencies of the execution, and other possible ways to build the final data structure. However, these other ways need more maximum working space (at some step) than the way listed in this table.

Lemma 5.8. *The data structure of Theorem 5.1 can compute* $\mathsf{L}_T[p]$ *in constant time with* $O(\log n)$ *bits of additional working space for each* p *with* $0 \leq p \leq n-1$.

Proof. Suppose that we have the data structure D of Theorem 5.1 and want to know $\mathsf{L}_T[p]$. We query D with the interval $[p, p]$ to retrieve *one* SUS for the query interval $[p, p]$ in constant time. This can be achieved by stopping the retrieval after the first SUS $[i, j] \in \mathsf{SUS}_T([p, p])$ has been reported. Since all SUSs for $[p, p]$ have the same length, $\mathsf{L}_T[p] = j - i + 1$. The additional working space is $O(\log n)$ bits. □

Lemma 5.8 allows us to compute LD_T in $O(n)$ time, which we represent as an integer array with bit width two, thus using $2n$ bits of space. In the following, we build a compressed rank/select data structure on LD_T. This data structure is a self-index such that we no longer need to keep LD_T in memory. With LD_T we can access L_T, as can be seen by the following lemma:

Lemma 5.9. *There exists a data structure of size* $\lceil n \log_2 3 \rceil + o(n)$ *bits that can access* $\mathsf{L}_T[i]$, *and can compute* $predneq_{\mathsf{L}_T}(i)$ *and* $succneq_{\mathsf{L}_T}(i)$ *in constant time for each position* i *with* $0 \leq i \leq n-1$. *Given* MB_T *and* ME_T, *the data structure can be constructed in* $O(n)$ *time using*

$2n + \max\{\lceil n \log_2 3 \rceil, 2m\} + o(n)$ *bits of total working space, which includes the space for this data structure.*

Proof. The following equations hold for every text position i with $0 \leq i \leq n-1$:

$$\begin{aligned} \mathsf{L}_T[i] &= \mathsf{L}_T[0] + rank_{\mathsf{LD}_T}(1, i) - rank_{\mathsf{LD}_T}(-1, i), \\ predneq_{\mathsf{L}_T}(i) &= \max\{select_{\mathsf{LD}_T}(c, rank_{\mathsf{LD}_T}(c, i)) - 1 \mid c \in \{-1, 1\}\}, \\ succneq_{\mathsf{L}_T}(i) &= \min\{select_{\mathsf{LD}_T}(c, rank_{\mathsf{LD}_T}(c, i) + 1) \mid c \in \{-1, 1\}\}. \end{aligned}$$

We can compute the value of $\mathsf{L}_T[0]$ and LD_T with Lemma 5.8. With a rank/select data structure on LD_T we can compute the above functions. Such a data structure is the Huffman-shaped wavelet tree [41]. This data structure can be constructed in linear time and takes $\lceil n \log_2 3 \rceil + o(n)$ bits of space, since the possible number of different values in LD_T is three. Therefore, it can also provide answers to rank/select queries in constant time. □

Finally, we show how to compute B_T:

Lemma 5.10. *There exists a data structure of size $n+o(n)$ bits that can compute $succ1pos_{\mathsf{B}_T}(i)$ and $pred1pos_{\mathsf{B}_T}(i)$ in constant time for each position $0 \leq i \leq n-1$. Given MB_T and ME_T, this data structure can be constructed in $O(n)$ time using $3n + \lceil n \log_2 3 \rceil + o(n)$ bits of total working space including the space for this data structure.*

Proof. Our idea is to compute B_T since the following equations hold for every text position i with $0 \leq i \leq n-1$ (cf. BIT_Z in Section 2.2):

$$\begin{aligned} pred1pos_{\mathsf{B}_T}(i) &= \begin{cases} i & \text{if } \mathsf{B}_T[i] = 1, \\ select_{\mathsf{B}_T}(1, rank_{\mathsf{B}_T}(1, i)) & \text{if } \mathsf{B}_T[i] = 0. \end{cases} \\ succ1pos_{\mathsf{B}_T}(i) &= \begin{cases} i & \text{if } \mathsf{B}_T[i] = 1, \\ select_{\mathsf{B}_T}(1, rank_{\mathsf{B}_T}(1, i) + 1) & \text{if } \mathsf{B}_T[i] = 0. \end{cases} \end{aligned}$$

In the following we show how to compute B_T from MB_T and ME_T in linear time with linear number of bits of working space. Let $b_i = select_{\mathsf{MB}_T}(1, i)$ and $e_i = select_{\mathsf{ME}_T}(1, i)$ be the starting position and the ending position of the i-th MUS respectively, for each $0 \leq i \leq m-1$. Given $x_i = \mathsf{RMQ}_{\mathsf{L}_T}(b_i, e_i)$, $\mathsf{L}_T[x_i] \leq e_i - b_i + 1$ since $b_i \leq x_i \leq e_i$ and $[b_i, e_i]$ is unique. If $\mathsf{L}_T[x_i] < e_i - b_i + 1$, there is no position p with $cover([b_i, e_i], p) \in \mathsf{SUS}_T(p)$, i.e., $[b_i, e_i]$ is a meaningless MUS. Otherwise ($\mathsf{L}_T[x_i] = e_i - b_i + 1$), $cover([b_i, e_i], x_i) = [b_i, e_i] \in \mathsf{SUS}_T(x_i)$,

i.e., $[b_i, e_i]$ is a meaningful MUS. Hence, it can be detected in constant time whether a MUS is meaningful by an RMQ query on L_T. We can compute the compact representation of L_T described in Lemma 5.9. The data structure takes $\lceil n \log_2 3 \rceil + o(n)$ bits and can be constructed with $2n + \max\{\lceil n \log_2 3 \rceil, 2m\} + o(n)$ bits of total working space. Subsequently, we endow it with the RMQ data structure of Lemma 2.2 in $O(n)$ time using $2n + o(n)$ bits of space. Therefore, the computing time of B_T is $O(n)$ and the working space is, aside from the space for MB_T and ME_T, $3n + \lceil n \log_2 3 \rceil + o(n)$ bits, including the space for B_T. Finally, we can endow B_T with rank/select data structures, which allows us to compute each of the above two functions $pred1pos_{\mathsf{B}_T}$ and $succ1pos_{\mathsf{B}_T}$ in constant time. □

Actually, having MB_T and ME_T available, we can simulate an access to $\mathsf{L}_T[i]$ in constant time with Lemma 5.8 by using the RmQ data structure on MUSlen_T. This allows us to compute the RMQ data structure on L_T directly without the need for computing B_T in the first place, i.e., we can replace the working space of Lemma 5.10 with $2n + 2m + o(n)$ additional bits of working space. However, since our final data structure needs LD_T, computing B_T before LD_T would require more working space in the end than the other way around, since we no longer need the RmQ data structure on MUSlen_T after having built the rank/select data structure of Lemma 5.9.

Before stating our final theorem, we need a property for meaningful MUSs:

Lemma 5.11. *On the one hand, $cover([s_i, e_i], p) \in \mathsf{SUS}_T(p)$ for every meaningful MUS $[s_i, e_i]$ starting with or after the leftmost MUS for p and starting before or with the rightmost MUS for p. On the other hand, each element (i.e., an interval) of $\mathsf{SUS}_T(p)$ starting with or after the leftmost MUS for p and starting before or with the rightmost MUS for p contains exactly one distinct MUS.*

Proof. The first part is shown by Tsuruta et al. [55, Lemma 3]. The second part is due to Lemma 5.1. □

Theorem 5.2. *For the point SUS problem, there exists a data structure of size $n + \lceil n \log_2 3 \rceil + o(n)$ bits that can answer a point SUS query in $O(k)$ time, where k is the number of SUSs of T for the respective query point. Given MB_T and ME_T, the data structure can be constructed in $O(n)$ time using $3n + \lceil n \log_2 3 \rceil + o(n)$ bits of total working space, which includes the space for this data structure.*

Proof. Let p be a query position, and suppose that the number of SUSs for p is k. Like the MUSs in Section 5.1, we rank the SUSs for p by their starting positions. Let $[s_j, e_j]$ be the j-th

data structure	Lemma	Time: Access $\pi_a(n)$	Time: Construction $\pi_c(n)$	Space in bits $\pi_s(n)$
succinct suffix tree	5.12	$O(\epsilon^{-1})$	$O(\epsilon^{-2}n)$	$(1+\epsilon)n\log n + O(n)$
compressed suffix tree	5.13	$O(\log_\sigma^\epsilon n)$	$O(n\log_\sigma^\epsilon n)$	$O(\epsilon^{-1}n\log\sigma)$
RLBWT	5.14	$O(\log\tilde{r}\log^{O(1)} n)$	$O(n\log\tilde{r}$ $+\tilde{r}\log^{O(1)} n)$	$\tilde{r}\log^{O(1)} n + O(n)$

Table 5.2: Efficient representations of $\mathcal{D}_T$ described in Lemma 5.2, i.e., data structures with access to $\mathsf{ISA}_T[i]$ and $\mathsf{LCP}_T[i]$. ϵ is a constant with $0 < \epsilon \leq 1$. $\tilde{r}$ denotes the number of single character runs in the BWT.

SUS for p with $0 \leq j \leq k-1$ such that $[s_0, e_0]$ and $[s_{k-1}, e_{k-1}]$ are the leftmost SUS and the rightmost SUS for p, respectively. If $s_0 = p$ then $[s_0, e_0] = [s_{k-1}, e_{k-1}]$, and thus the output consists of this single interval. Otherwise ($s_0 \neq p$), we can compute s_i iteratively from s_{i-1} by $s_i = select_{\mathsf{B}_T}(1, rank_{\mathsf{B}_T}(1, s_{i-1}) + 1)$ in constant time for each i with $1 \leq i \leq k-2$, allowing us to answer the query in time linear to the number of SUSs. As k is not known in advance, we stop the iteration whenever we computed an s_i that is larger than the starting position of the rightmost SUS for p. A detailed analysis of the claimed working space is given in Table 5.1. □

Corollary 5.3. *The data structure of Theorem 5.2 can compute the number of SUSs for a query position in constant time.*

Proof. Let $[s_l, e_l]$ and $[s_r, e_r]$ be the leftmost and the rightmost SUS for a given query position, respectively. All MUSs starting between s_l and s_r (excluding s_l and s_r) are SUSs for this query position. Let k' be their number. Therefore, the number we want to output is $k = k' + 2$. With Lemmas 5.6 and 5.7, we can find $[s_l, e_l]$ and $[s_r, e_r]$ in constant time. Further, we can compute k' in constant time since $k' = rank_{\mathsf{B}_T}(1, s_r - 1) - rank_{\mathsf{B}_T}(1, s_l)$. □

5.4 Auxiliary Data Structure

In the following, we present three different representations of the data structure required by Lemma 5.2. See Table 5.2 for a juxtaposition of their characteristics. Also, we present an algorithm for computing MUSs by using $\mathsf{RankSucc}_T$ and a succinct representation of PLCP_T. In this subsection, we assume that Σ is an integer alphabet of size $\sigma = n^{O(1)}$.

Lemma 5.12 ([19, Section 2.2.3])**.** *Given a constant ϵ with $0 < \epsilon \leq 1$, there is a data structure that can access $\mathsf{ISA}_T[i]$ and $\mathsf{LCP}_T[i]$ in $\pi_a(n) = O(\epsilon^{-1})$ time using $\pi_s(n) = (1+\epsilon)n \log n + O(n)$ bits of working space. It can be constructed within this working space in $\pi_c(n) = O(\epsilon^{-2}n)$ time.*

Lemma 5.13. *Given a constant ϵ with $0 < \epsilon \leq 1$, there is a data structure that can access $\mathsf{ISA}_T[i]$ and $\mathsf{LCP}_T[i]$ in $\pi_a(n) = O(\log_\sigma^\epsilon n)$ time using $\pi_s(n) = O(\epsilon^{-1} n \log \sigma)$ bits of working space. It can be constructed within this working space in $\pi_c(n) = O(n \log_\sigma^\epsilon n)$ time.*

Proof. We use the compressed suffix tree presented in [45]. While the succinct suffix tree naturally supports the required access queries, the compressed suffix tree needs to be enhanced with a sampling data structure to support access to $\mathsf{ISA}_T[i]$ and $\mathsf{LCP}_T[i]$. In detail, it provides access to the PLCP array PLCP_T. Our idea is to provide access to SA_T and ISA_T in $O(\log^\epsilon n)$ time. For the former, we use a sampling of SA_T with $O(\epsilon^{-1} n \log \sigma)$ bits to obtain access to SA_T with $O(\log_\sigma^\epsilon n)$ time [23, Section 3.2]. For the latter, having this SA_T representation, we create a representation of ISA_T using $O(\epsilon^{-1} n \log \sigma)$ space in $O(n \log_\sigma^\epsilon n)$ time [46]. This representation can access $\mathsf{ISA}_T[i]$ in $O(\log^\epsilon n)$ time. □

Lemma 5.14. *There is a data structure that can access $\mathsf{ISA}_T[i]$ and $\mathsf{LCP}_T[i]$ in $\pi_a(n) = O(\log \tilde{r} \log^{O(1)} n)$ time using $\pi_s(n) = \tilde{r} \log^{O(1)} n + O(n)$ bits of working space. It can be constructed within this working space in $\pi_c(n) = O(n + \tilde{r} \log^{O(1)} n)$ time.*

Proof. Like in the proof of Lemma 5.13, we provide access to LCP_T by PLCP_T and SA_T: First, we compute the run-length encoded BWT in $O(n \log \tilde{r})$ time stored in $O(\tilde{r} \log n)$ bits of space [47]. Next, we build a data structure answering $\mathsf{SA}_T[i]$ and $\mathsf{ISA}_T[i]$ in $O(\log^{O(1)} n \log \tilde{r})$ time while using $O(\tilde{r} \log n + n)$ bits of space [33, Theorem 5.1]. It can be constructed in $O(n + \tilde{r} \log^{O(1)} n)$ time with $O(n + \tilde{r} \log^{O(1)} n)$ bits of working space.

□

Lemma 5.15. *We can compute MB_T and ME_T in $O(n)$ time using $n \log n + O(\sigma \log n) + 4n + o(n)$ bits of total working space including the space for MB_T and ME_T.*

Proof. First, we construct $\mathsf{RankPred}_T$ from T using additional $O(\sigma \lg n)$ bits of working space [22]. Our second target is the succinct representation of the PLCP array $succPLCP_T$ of Sadakane [51] using $2n + o(n)$ bits of space while allowing constant time random access to each PLCP value. We can compute it from $\mathsf{RankPred}_T$ [32]. After computing $succPLCP_T$, we delete $\mathsf{RankPred}_T$. Subsequently, we construct $\mathsf{RankSucc}_T$ from T similarly to Goto et al. [22][3]. Finally, we obtain $succPLCP_T$ and $\mathsf{RankSucc}_T$. All the steps can be executed in $O(n)$ time.

In the following, we describe an algorithm for computing all MUSs of T by using PLCP_T (in the succinct representation $succPLCP_T$) and $\mathsf{RankSucc}_T$. As described in the proof of Lemma 5.2, if we can compute $\ell_i = \max\{\mathsf{LCP}_T[\mathsf{ISA}_T[i]], \mathsf{LCP}_T[\mathsf{ISA}_T[i]+1]\}$ in $\pi'_a(n)$ time for each text position i with $1 \leq i \leq n$, then we can compute MB_T and ME_T in $O(n \cdot \pi'_a(n))$ time. Actually, we can achieve that $\pi'_a(n) = O(1)$ by using the fact that $\ell_i = \max\{\mathsf{PLCP}_T[i], \mathsf{PLCP}_T[\mathsf{RankSucc}_T[i]]\}$. Therefore, we can compute MB_T and ME_T in $O(n)$ time. Also, the total working space for the above procedure is $n \log n + O(\sigma \lg n) + 4n + o(n)$ bits including the space for MB_T and ME_T. □

5.5 Conclusions

In this chapter, we proposed compact data structures for the interval SUS problem and the point SUS problem. Our data structure is the first data structure of size $O(n)$ bits for the interval SUS problem (resp. the point SUS problem). On the one hand, for the interval SUS problem, we proposed a data structure of size $2n+2m+o(n)$ bits answering a query in output-sensitive $O(k)$ time, where n is the length of the input string, m is the number of MUSs in the input string, and k is the number of returned SUSs. On the other hand, for the point SUS problem, we proposed a data structure of size $\lceil(\log_2 3+1)n\rceil + o(n)$ bits answering a query in the same output-sensitive time. The construction time and the working space of each data structures depends on the complexity of simulating the inverse suffix array and the LCP array of the input string. For example, by using the succinct suffix tree [19], we can achieve $O(n)$ time and $2n \log n + O(n)$ bits of working space to construct our data structures.

[3]They proposed a construction algorithm only for $\mathsf{RankPred}_T$. However, we can modify the algorithm to construct $\mathsf{RankSucc}_T$. Their algorithm simulates a suffix array construction algorithm, but uses $O(\sigma \lg n)$ pointers to positions in $\mathsf{RankPred}_T$ at which new values are inserted. By these pointers, they represent $\mathsf{RankPred}_T$ as 2σ linked lists. In [22, Step 4 in Section 4.2.2], they show how to invert these lists. Consequently, we can use this step to invert all lists after all elements have been inserted.

5.5.1 Open Problems

We are unaware of an algorithm computing $\mathsf{RankPred}_T$ or $\mathsf{RankSucc}_T$ from the text T in-place for integer alphabets. Since it is possible to store $\mathsf{RankPred}_T$ in unary with the same representation as the succinct PLCP representation [51] in $O(r)$ space [35], where r is the number of runs in the BWT, we wonder whether we can compute $\mathsf{RankPred}_T$ in compressed space. A possibility seems to adapt the in-place suffix array construction algorithm of Li et al. [38].

As future work, we want to extend our algorithm to compute SUSs with k edits (or k mismatches). A SUS with k edits is a substring that is unique even when changing k arbitrary characters (allowing deletions, insertions and character exchanges). Similarly, a SUS with k mismatches is unique even when exchanging k arbitrary characters. It is known that SUSs with k edits (resp. mismatches) contain SUSs with $k-1$ edits (resp. mismatches), where SUSs with no edits (resp. no mismatches) are the ordinary SUSs.

Chapter 6

Computing Minimal Unique Substrings for a Semi-Dynamic String

In this chapter, we tackle the problem of computing minimal unique substrings in a semi-dynamic string. As we mentioned in Chapter 2, our semi-dynamic setting allows us to append a character to the right-end of the input string or delete the leftmost character from the string. Alternating these two operations immediately realizes the *sliding-window* operation. Hence, in what follows, we will regard a problem in semi-dynamic strings as a problem in the sliding window model. A typical application to the sliding window model is data compression; examples are the famous Lempel-Ziv 77 (the original version) [58] and PPM [12]. Recently, Crochemore et al. [13] introduced the problem of computing *Minimal Absent Words* for a sliding window, and proposed an $O(n\sigma)$-time and $O(d\sigma)$-space algorithm using suffix trees for a sliding window where d is the window size and σ is the alphabet size.

We begin with combinatorial results on MUSs for a sliding window. Namely, we show that the number of MUSs that are added or deleted by one slide of the window is always constant (Section 6.2). We then present the first efficient algorithm that maintains the set of MUSs for a sliding window of length d over a string of length n in a total of $O(n \log \sigma')$ time and $O(d)$ working space where $\sigma' \leq d$ is the maximum number of distinct characters in every window (Section 6.3). Our main algorithmic tool is the suffix tree for a sliding window that requires $O(d)$ space and can be maintained in $O(n \log \sigma')$ time [17, 37, 52]. Our algorithm for computing MUSs for a sliding window is built on our combinatorial results, and it keeps track of three different loci over the suffix tree, all of which can be maintained in $O(\log \sigma')$ amortized time per each sliding step.

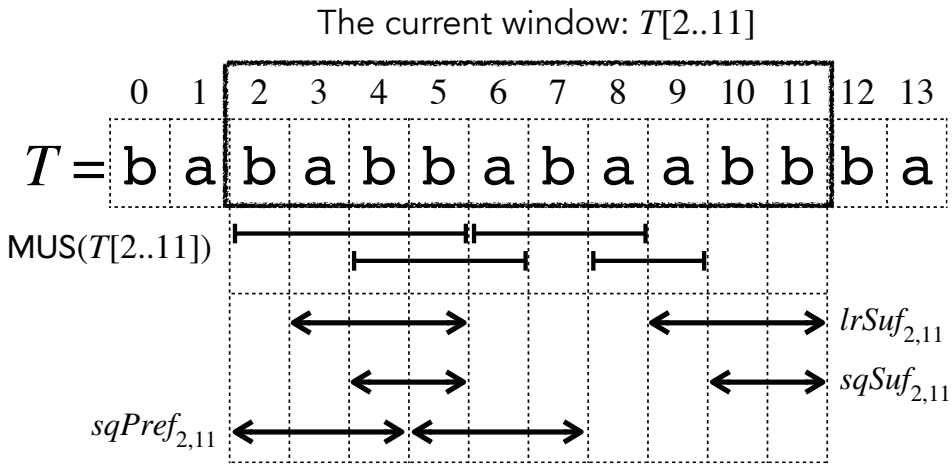

Figure 6.1: String $T =$ `bababbabaabbba` of length 14 and its substrings $lrSuf_{2,11}$, $sqSuf_{2,11}$, and $sqPref_{2,11}$ for the current window $T[2..11]$.

We emphasize that our algorithms work for any valid[1] sequence of windows of arbitrary lengths. However, for simplicity, we consider the case where a fix-sized window of length d shifts to the right one-by-one throughout the rest of this chapter (and the next chapter as well).

6.1 Preliminaries

For any $0 \leq i \leq j \leq |T| - 1$, $lrSuf_{i,j}$ denotes the longest repeating suffix of $T[i..j]$, $sqSuf_{i,j}$ denotes the shortest quasi-unique suffix of $T[i..j]$, and $sqPref_{i,j}$ denotes the shortest quasi-unique prefix of $T[i..j]$. While $lrSuf_{i,j}$ can be the empty string, both $sqSuf_{i,j}$ and $sqPref_{i,j}$ are always non-empty strings for any i, j with $0 \leq i \leq j \leq |T| - 1$. See Fig. 6.1 for examples.

See Fig. 6.1 for examples of MUSs.

In this chapter, we tackle the problem of computing MUSs for a sliding window of fixed length d over a given string T, formalized as follows:

Input: String T of length n and positive integer d ($< n$).

Output: $\mathsf{MUS}_{T[i..i+d-1]}$ for all $0 \leq i \leq n - d$.

This chapter deals with the problem of computing MUSs for a sliding window

[1] C.f., the definition of sliding windows in Section 2.1

6.2 Combinatorial Results on MUSs for a Sliding Window

Throughout this section, we consider positions i and j with $0 \leq i \leq j \leq n-1$ such that $T[i..j]$ denotes the sliding window for the i-th position over the input string T. The following arguments hold for *any* values of i and j, and hence, they will be useful for sliding windows of any length d. The next lemmas are useful for analyzing combinatorial properties of MUSs and for designing an efficient algorithm for computing MUSs for a sliding window.

Lemma 6.1. *The following three statements are equivalent:*

(1) $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$,

(2) $\#occ_{T[i..j]}(lrSuf_{i,j}) = 2$, *and*

(3) $\#occ_{T[i..j]}(sqSuf_{i,j}) = 2$.

Proof. **(1) ⇒ (2) and (3):** Since $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$, $sqSuf_{i,j}$ is a suffix of $lrSuf_{i,j}$ and thus $\#occ_{T[i..j]}(sqSuf_{i,j}) \geq \#occ_{T[i..j]}(lrSuf_{i,j})$. By the definitions of $sqSuf_{i,j}$ and $lrSuf_{i,j}$, $\#occ_{T[i..j]}(sqSuf_{i,j}) \leq 2$ and $\#occ_{T[i..j]}(lrSuf_{i,j}) \geq 2$. Thus $\#occ_{T[i..j]}(lrSuf_{i,j}) = \#occ_{T[i..j]}(sqSuf_{i,j}) = 2$.
(2) ⇒ (1): Since $\#occ_{T[i..j]}(lrSuf_{i,j}) = 2$, the shortest suffix $sqSuf_{i,j}$ of $T[i..j]$ that occurs at most twice in $T[i..j]$ cannot be longer than $lrSuf_{i,j}$, i.e., $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$.
(3) ⇒ (1): Since $\#occ_{T[i..j]}(sqSuf_{i,j}) = 2$, the longest suffix $lrSuf_{i,j}$ of $T[i..j]$ that occurs at least twice in $T[i..j]$ is at least as long as $sqSuf_{i,j}$, i.e., $|lrSuf_{i,j}| \geq |sqSuf_{i,j}|$. □

Fig. 6.1 shows a concrete example where (1) of Lemma 6.1 holds (and hence both (2) and (3) also hold.)

Lemma 6.2. $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}| + 1$.

Proof. Assume on the contrary that $|lrSuf_{i,j+1}| > |lrSuf_{i,j}| + 1$. By the definition of $lrSuf_{i,j+1}$, $lrSuf_{i,j+1} = T[j+2-|lrSuf_{i,j+1}|..j+1]$ occurs at least twice in $T[i..j+1]$. Hence, $T[j+2-|lrSuf_{i,j+1}|..j]$ which is a proper prefix of $lrSuf_{i,j+1}$ also occurs at least twice in $T[i..j]$. In addition, $lrSuf_{i,j} = T[j+2-|lrSuf_{i,j}|..j]$ is a proper suffix of $T[j+2-|lrSuf_{i,j+1}|..j]$ since $|lrSuf_{i,j+1}| > |lrSuf_{i,j}| + 1$. However, this contradicts the definition of $lrSuf_{i,j}$. Therefore, $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}| + 1$. □

6.2.1 Changes to MUSs When Appending a Character to the Right

In this subsection, we consider an operation that slides the right-end of the current window $T[i..j]$ with one character by appending the next character $T[j+1]$ to $T[i..j]$. We use the following observation.

Observation 6.1. *For any non-empty substring s of $T[i..j]$, $\#occ_{T[i..j+1]}(s) \leq \#occ_{T[i..j]}(s)$ $+1$. Also, $\#occ_{T[i..j+1]}(s) = \#occ_{T[i..j]}(s) + 1$ if and only if s is a suffix of $T[i..j+1]$.*

MUSs to be Deleted When Appending a Character to the Right

Due to Observation 6.1, we obtain Lemma 6.3 which describes MUSs to be deleted when a new character $T[j+1]$ is appended to the current window $T[i..j]$.

Lemma 6.3. *For any $[s,t]$ with $i \leq s < t \leq j$, $[s,t] \in \mathsf{MUS}_{T[i..j]}$ and $[s,t] \notin \mathsf{MUS}_{T[i..j+1]}$ if and only if $T[s..t] = sqSuf_{i,j+1}$ and $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$.*

Proof. ($\Rightarrow$) Let $w = T[s..t]$. Since $[s,t] \in \mathsf{MUS}_{T[i..j]}$ and $[s,t] \notin \mathsf{MUS}_{T[i..j+1]}$, $\#occ_{T[i..j]}(w) = 1$ and $\#occ_{T[i..j+1]}(w) \geq 2$. It follows from Observation 6.1 that $\#occ_{T[i..j+1]}(w) = 2$ and w is a suffix of $T[i..j+1]$. If we assume that w is a proper suffix of $sqSuf_{i,j+1}$, then $\#occ_{T[i..j+1]}(w) \geq 3$ by the definition of $sqSuf_{i,j+1}$, but this contradicts with $\#occ_{T[i..j+1]}(w) = 2$. If we assume that $sqSuf_{i,j+1}$ is a proper suffix of w, then $\#occ_{T[i..j]}(sqSuf_{i,j+1}) \geq \#occ_{T[i..j]}(T[s+1..t]) \geq 2$. Also, $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = \#occ_{T[i..j]}(sqSuf_{i,j+1}) + 1 \geq 3$ by Observation 6.1, but this contradicts the definition of $sqSuf_{i,j+1}$. Therefore, $w = sqSuf_{i,j+1}$. Moreover, $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$ since $w = sqSuf_{i,j+1}$ is a substring of $T[i..j]$.
($\Leftarrow$) Since $w = T[s..t]$ is a suffix of $T[i..j+1]$ and $\#occ_{T[i..j+1]}(w) = 2$, w is unique in $T[i..j]$. By the definition of $sqSuf_{j+1}$, a proper suffix $w[1..] = T[s+1..t]$ of $w = sqSuf_{i,j+1}$ occurs at least three times in $T[i..j+1]$, i.e., $T[s+1..t]$ is repeating in $T[i..j]$ (see also Fig. 6.2 for illustration). Also, a prefix $w[0..|w|-2] = T[s..t-1]$ of $w = sqSuf_{i,j+1}$ is clearly repeating in $T[i..j]$. Therefore, $w = T[s..t]$ is a MUS of $T[i..j]$ and is not a MUS of $T[i..j+1]$. □

By Lemma 6.3, at most one MUS can be deleted when appending $T[j+1]$ to the current window $T[i..j]$, and such a deleted MUS must be $sqSuf_{i,j+1}$.

MUSs to be Added When Appending a Character to the Right

First, we consider a MUS to be added when appending $T[j+1]$ to $T[i..j]$, which is a suffix of $T[i..j+1]$. The next observation follows from the definition of $lrSuf_{i,j}$:

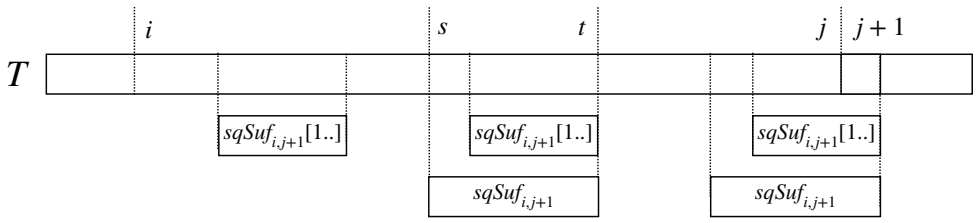

Figure 6.2: Illustration for the case where $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$. In this case, $T[s..t] = sqSuf_{i,j+1}$ is unique in $T[i..j]$ and $T[s+1..t]$ is repeating in $T[i..j]$.

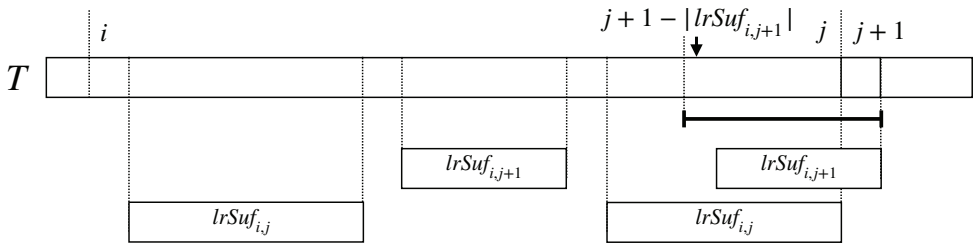

Figure 6.3: Illustration for the case where $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}|$. In this case, $T[j+1-|lrSuf_{i,j+1}|..j+1]$ is a MUS of $T[i..j+1]$.

Observation 6.2. *If $[s,j] \in \mathsf{MUS}_{T[i..j]}$, then $s = j - |lrSuf_{i,j}|$. Namely, if there is a MUS of $T[i..j]$ that is a suffix of $T[i..j]$, then it must be the suffix of $T[i..j]$ that is exactly one character longer than $lrSuf_{i,j}$.*

Lemma 6.4. *The interval $[j+1-\ell, j+1] \in \mathsf{MUS}_{T[i..j+1]}$ if and only if $T[j+1-\ell..j+1] = \alpha^{\ell+1}$ or $\ell \leq |lrSuf_{i,j}|$, where $\ell = |lrSuf_{i,j+1}|$ and $\alpha = T[j+1]$.*

Proof. ($\Rightarrow$) Assume on the contrary that $T[j+1-\ell..j+1] \neq \alpha^{\ell+1}$ and $\ell > |lrSuf_{i,j}|$. By the assumptions and Lemma 6.2, $|lrSuf_{i,j}| = \ell - 1$, and thus, $T[j-|lrSuf_{i,j}|..j] = T[j+1-\ell..j]$. Since $T[j+1-\ell..j+1]$ is a MUS of $T[i..j+1]$, $T[j+1-\ell..j] = T[j-|lrSuf_{i,j}|..j]$ occurs at least twice in $T[i..j+1]$. On the other hand, $T[j-|lrSuf_{i,j}|..j]$ is unique in $T[i..j]$ by the definition of $lrSuf_{i,j}$, hence $T[j-|lrSuf_{i,j}|..j]$ occurs in $T[i..j+1]$ as a suffix of $T[i..j+1]$. Consequently, we have $T[j-|lrSuf_{i,j}|..j] = T[j+1-|lrSuf_{i,j}|..j+1]$, i.e., $T[j-\ell..j] = T[j+1-\ell..j+1] = \alpha^{\ell+1}$ with $\alpha = T[j+1]$, a contradiction.
($\Leftarrow$) By the definition of $lrSuf_{i,j+1}$, $T[j+2-\ell..j+1] = lrSuf_{i,j+1}$ is repeating in $T[i..j+1]$, and $T[j+1-\ell..j+1]$ is unique in $T[i..j+1]$. Now it suffices to show $T[j+1-\ell..j]$ is repeating in $T[i..j+1]$. If $T[j+1-\ell..j+1] = \alpha^{\ell+1}$, then clearly $T[j+1-\ell..j] = \alpha^{\ell}$ is repeating in $T[i..j+1]$. If $\ell \leq |lrSuf_{i,j}|$, then $T[j+1-\ell..j]$ is a suffix of $T[j+1-|lrSuf_{i,j}|..j]$ (see Fig. 6.3). Thus $\#occ_{T[i..j+1]}(T[j+1-\ell..j]) \geq \#occ_{T[i..j]}(T[j+1-\ell..j]) \geq \#occ_{T[i..j]}(T[j+1-|lrSuf_{i,j}|..j]) \geq$

2. □

Next, we consider MUSs to be added when appending $T[j+1]$ to $T[i..j]$, which are *not* suffixes of $T[i..j+1]$.

Lemma 6.5. *For each* $[s,t] \in \mathsf{MUS}_{T[i..j+1]}$ *with* $t \neq j+1$, *if* $[s,t] \notin \mathsf{MUS}_{T[i..j]}$ *then* $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$ *and* $sqSuf_{i,j+1}$ *is a proper substring of* $T[s..t]$.

Proof. Since $t \neq j+1$, $T[s..t]$ is not a suffix of $T[i..j+1]$. Moreover, since $[s,t] \in \mathsf{MUS}_{T[i..j+1]}$, $T[s..t]$ is unique in $T[i..j]$. Since $T[s..t]$ is not a MUS of $T[i..j]$, there exists a MUS u of $T[i..j]$ which is a proper substring of $T[s..t]$. Assume on the contrary that $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 1$ or $u \neq sqSuf_{i,j+1}$. Then, it follows from Lemma 6.3 that u is a MUS of $T[i..j+1]$. However, this contradicts that $[s,t] \in \mathsf{MUS}_{T[i..j+1]}$. Therefore, $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$ and $u = sqSuf_{i,j+1}$ is a proper substring of $T[s..t]$. □

Namely, a MUS which is not a suffix is added by appending one character only if there is a MUS to be deleted by the same operation. Moreover, such added MUSs must contain the deleted MUS.

Lemma 6.6. *If* $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$, *then there are three integers* p_l, p_s, q *such that* $i \leq p_l \leq p_s \leq q < j+1$, $T[p_s..q] = sqSuf_{i,j+1}$ *and* $T[p_l..q] = lrSuf_{i,j+1}$. *Also, the following propositions hold:*

(a) *If there is no MUS of* $T[i..j]$ *ending at* $q+1$, *then* $[p_s, q+1] \in \mathsf{MUS}_{T[i..j+1]}$.

(b) *If there is no MUS of* $T[i..j]$ *starting at* $p_l - 1$ *and* $p_l \geq i+1$, *then* $[p_l - 1, q] \in \mathsf{MUS}_{T[i..j+1]}$.

Proof. Since $\#occ_{T[i..j+1]}(sqSuf_{i,j+1}) = 2$, it follows from Lemma 6.1 that $sqSuf_{i,j+1}$ is a suffix of $lrSuf_{i,j+1}$ and $\#occ_{T[i..j+1]}(lrSuf_{i,j+1}) = 2$. Hence, the ending positions of the occurrence of $sqSuf_{i,j+1}$ in $T[i..j]$ and that of $lrSuf_{i,j+1}$ in $T[i..j]$ are the same (see Fig. 6.4).

Next, we consider MUSs to be added.

(a) For the sake of contradiction, assume that $T[p_s..q+1]$ is repeating in $T[i..j+1]$. Then $\#occ_{T[i..j+1]}(T[p_s..q]) \geq 3$, and it contradicts the definition of $sqSuf_{i,j+1}$ $(= T[p_s..q])$. Hence, $T[p_s..q+1]$ is unique in $T[i..j+1]$. Also, it is clear that $T[p_s..q] = sqSuf_{i,j+1}$ is repeating in $T[i..j+1]$. In addition, $T[p_s+1..q+1]$ is repeating in $T[i..j]$ since $[p_s, q] \in \mathsf{MUS}_{T[i..j]}$ (by Lemma 6.3) and there is no MUS of $T[i..j]$ ending at $q+1$. Thus, $T[p_s+1..q+1]$ is also repeating in $T[i..j+1]$. Therefore, $T[p_s..q+1]$ is a MUS of $T[i..j+1]$.

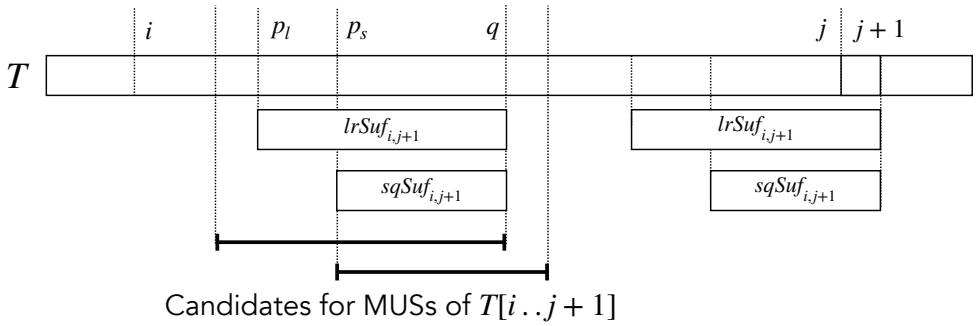

Figure 6.4: Illustration of the situation when $sqSuf_{i,j+1}$ is repeating in $T[i..j+1]$. In this situation, $[p_l - 1, q]$ and $[p_s, q+1]$ are the only candidates for MUSs in $\mathsf{MUS}_{T[i..j+1]} \setminus \mathsf{MUS}_{T[i..j]}$ each of which is not a suffix of $T[i..j+1]$.

(b) For the sake of contradiction, assume that $T[p_l - 1..q]$ is repeating in $T[i..j+1]$. Then $T[p_l - 1] = T[j + 1 - |lrSuf_{i,j+1}|]$ or $\#occ_{T[i..j+1]}(T[p_l..q]) \geq 3$. If $T[p_l - 1] = T[j + 1 - |lrSuf_{i,j+1}|]$, it contradicts the definition of $lrSuf_{i,j+1}$. On the other hand, if $\#occ_{T[i..j+1]}(T[p_l..q]) \geq 3$, it contradicts $\#occ_{T[i..j+1]}(lrSuf_{i,j+1}) = 2$. Thus, $T[p_l - 1..q]$ is unique in $T[i..j+1]$. Also, it is clear that $T[p_l..q] = lrSuf_{i,j+1}$ is repeating in $T[i..j+1]$. In addition, $T[p_l - 1..q - 1]$ is repeating in $T[i..j]$, since $[p_s, q] \in \mathsf{MUS}_{T[i..j]}$ (by Lemma 6.3), $T[p_l..q-1]$ is repeating in $T[i..j]$, and there is no MUS of $T[i..j]$ starting at $p_l - 1$. Thus, $T[p_l - 1..q - 1]$ is repeating in $T[i..j+1]$. Therefore, $T[p_l - 1..q]$ is a MUS of $T[i..j+1]$.

□

Now we have the main result of this subsection:

Theorem 6.1. *For any* $0 \leq i \leq j < n - 1$, $|\mathsf{MUS}_{T[i..j+1]} \triangle \mathsf{MUS}_{T[i..j]}| \leq 4$ *and* $-1 \leq |\mathsf{MUS}_{T[i..j+1]}| - |\mathsf{MUS}_{T[i..j]}| \leq 2$. *Furthermore, these bounds are tight for any* σ, i, j *with* $\sigma \geq 3$, $0 \leq i \leq j < n - 1$, *and* $j - i + 1 \geq 5$.

Proof. First, we show that $|\mathsf{MUS}_{T[i..j+1]} \triangle \mathsf{MUS}_{T[i..j]}| \leq 4$. By Lemma 6.3, $|\mathsf{MUS}_{T[i..j]} \setminus \mathsf{MUS}_{T[i..j+1]}| \leq 1$. By Observation 6.2 and Lemma 6.6, $|\mathsf{MUS}_{T[i..j+1]} \setminus \mathsf{MUS}_{T[i..j]}| \leq 3$. Thus, $|\mathsf{MUS}_{T[i..j+1]} \triangle \mathsf{MUS}_{T[i..j]}| = |\mathsf{MUS}_{T[i..j+1]} \setminus \mathsf{MUS}_{T[i..j]}| + |\mathsf{MUS}_{T[i..j]} \setminus \mathsf{MUS}_{T[i..j+1]}| \leq 4$. Also, we show that the upper bound is tight if $\sigma \geq 3$. For an integer $k \geq 2$, we consider two strings u and u' such that $u = \mathtt{a}^k\mathtt{bcc}$ of length $k + 3 \geq 5$ and $u' = u\mathtt{b} = \mathtt{a}^k\mathtt{bccb}$ of length $k + 4 \geq 6$. Then, $\mathsf{MUS}_u = \{[0, k-1], [k, k], [k+1, k+2]\}$ and $\mathsf{MUS}_{u'} = \{[0, k-1], [k-1, k], [k, k+1], [k+1, k+2], [k+2, k+3]\}$. Therefore, $|\mathsf{MUS}_{u'} \triangle \mathsf{MUS}_u| = 4$.

Next, we show that $-1 \leq |\mathsf{MUS}_{T[i..j+1]}| - |\mathsf{MUS}_{T[i..j]}| \leq 2$. By Lemma 6.3, it is clear that $-1 \leq |\mathsf{MUS}_{T[i..j+1]}| - |\mathsf{MUS}_{T[i..j]}|$. By Observation 6.2, the number of added MUSs which are suffixes of $T[i..j+1]$ is at most one. Also, by Lemma 6.6, the number of added MUSs which are not suffixes of $T[i..j+1]$ is at most two, however, if such an added MUS exists, exactly one MUS ($= sqSuf_{i,j+1}$) must be deleted (cf. Lemma 6.3 and Lemma 6.5). Therefore, $|\mathsf{MUS}_{T[i..j+1]}| - |\mathsf{MUS}_{T[i..j]}| \leq 2$. Also, we show that each bound is tight when $\sigma \geq 3$. We consider strings u and u' that are described in the case (a), and we then obtain $|\mathsf{MUS}_{u'}| - |\mathsf{MUS}_u| = 2$. On the other hand, for any integer ℓ with $\ell \geq 1$, we consider two strings v and v'; $v = \mathtt{a}^\ell\mathtt{bcac}$ of length $\ell + 4 \geq 5$ and $v = v\mathtt{a} = \mathtt{a}^\ell\mathtt{bcaca}$ of length $\ell + 5 \geq 6$. If $\ell = 1$, then $\mathsf{MUS}_v = \{[1,1],[2,3],[3,4]\}$, and $\mathsf{MUS}_{v'} = \{[1,1],[3,4]\}$. If $\ell \geq 2$, then $\mathsf{MUS}_v = \{[0,\ell-1],[\ell,\ell],[\ell+1,\ell+2],[\ell+2,\ell+4]\}$, and $\mathsf{MUS}_{v'} = \{[0,\ell-1],[\ell,\ell],[\ell+2,\ell+3]\}$. Therefore, $|\mathsf{MUS}_{v'}| - |\mathsf{MUS}_v| = -1$. □

6.2.2 Changes to MUSs When Deleting the Leftmost Character

In this subsection, we consider an operation that deletes the leftmost character $T[i-1]$ from $T[i-1..j]$. Basically, we can use symmetric arguments to the previous subsection where we considered appending a character to the right of the window.

Observation 6.3. *For each non-empty substring s of $T[i-1..j]$, $\#occ_{T[i-1..j]}(s) \leq \#occ_{T[i..j]}(s) + 1$. Also, $\#occ_{T[i-1..i]}(s) = \#occ_{T[i..j]}(s) + 1$ if and only if s is a prefix of $T[i-1..j]$.*

MUSs to be Added When Deleting the Leftmost Character

Lemma 6.7. *For any $i \leq s \leq t \leq j$, $[s,t] \notin \mathsf{MUS}_{T[i-1..j]}$ and $[s,t] \in \mathsf{MUS}_{T[i..j]}$ if and only if $T[s..t] = sqPref_{i-1,j}$ and $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 2$.*

Proof. Symmetric to the proof of Lemma 6.3. □

MUSs to be Deleted When Deleting the Leftmost Character

Next, we consider MUSs to be deleted by removing $T[i-1]$ from $T[i-1..j]$. If there is a MUS w of $T[i-1..j]$ which is a prefix of $T[i-1..j]$, clearly, w is not a MUS of $T[i..j]$. Then, we consider MUSs to be deleted each of which are *not* a prefix of $T[i-1..j]$.

Lemma 6.8. *For each $[s,t] \in \mathsf{MUS}_{T[i-1..j]}$ with $s \neq i-1$, if $[s,t] \notin \mathsf{MUS}_{T[i..j]}$ then $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 2$ and $sqPref_{i-1,j}$ is a proper substring of $T[s..t]$.*

Proof. Symmetric to the proof of Lemma 6.5. □

Namely, when deleting the leftmost character, a MUS which is not a prefix is deleted only if an added MUS exists. Moreover, such deleted MUSs must contains the added MUS.

Lemma 6.9. *If* $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 2$*, then following propositions hold:*

(a) *If there is a MUS* w *starting at* s *in* $T[i-1..j]$*,* w *is not a MUS of* $T[i..j]$*,*

(b) *If there is a MUS* w' *ending at* t *in* $T[i-1..j]$*,* w' *is not a MUS of* $T[i..j]$*,*

where $T[s..t] = sqPref_{i-1,j}$ *and* $s \neq i-1$*.*

Proof. Symmetric to the proof of Lemma 6.6. □

The main result of this subsection is the following:

Theorem 6.2. *For any* $0 < i \leq j \leq n-1$*,* $|\mathsf{MUS}_{T[i-1..j]} \triangle \mathsf{MUS}_{T[i..j]}| \leq 4$ *and* $-1 \leq |\mathsf{MUS}_{T[i-1..j]}| - |\mathsf{MUS}_{T[i..j]}| \leq 2$*. Furthermore, these bounds are tight for any* σ, i, j *with* $\sigma \geq 3$*,* $0 < i \leq j \leq n-1$*, and* $j - i + 1 \geq 5$*.*

Proof. Symmetric to the proof of Theorem 6.1. □

The next corollary is immediate from Theorem 6.1 and Theorem 6.2.

Corollary 6.1. *Let* $0 < d < n$*. For every* $0 \leq i \leq n-d-1$*,* $|\mathsf{MUS}_{T[i..i+d-1]} \triangle \mathsf{MUS}_{T[i+1..i+d]}| \in O(1)$*.*

6.3 Algorithm for computing MUSs for a Sliding Window

This section presents our algorithm for computing MUSs for a sliding window.

6.3.1 Updating Suffix Tree and its Three Loci

First, we introduce some additional notions. Since we use Ukkonen's algorithm [56] for updating the suffix tree when a new character $T[j+1]$ is appended to the right end of the window $T[i..j]$, we maintain the locus for $lrSuf_{i,j}$ as in [56]. Also, in order to compute the changes of MUSs, we use $sqSuf_{i,j}$ (cf. Lemma 6.3 and Lemma 6.6). Thus, we also maintain the locus for $sqSuf_{i,j}$.

The locus for $lrSuf_{i,j}$ (resp. $sqSuf_{i,j}$) in $\mathsf{STree}(T[i..j])$ is called the *primary active point* (resp. the *secondary active point*) and is denoted by $\mathsf{pp}_{i,j}$ (resp. $\mathsf{sp}_{i,j}$). Additionally, in order to maintain $\mathsf{sp}_{i,j}$ efficiently, we also maintain the locus for the longest suffix of $T[i..j]$ which occurs at least three times in $T[i..j]$. We call this locus the *tertiary active point* that is denoted by $\mathsf{tp}_{i,j}$. See Fig. 2.2 for concrete examples of these three loci in a suffix tree.

Appending One Character

When $T[i..j]$ is the empty string (the base case, where $i = 0$ and $j = -1$), we set all the three active points $\langle root, 0\rangle$. Then we increase j, and the suffix tree grows in an online manner until $j = d - 1$ using Ukkonen's algorithm. Then, for each $j > d - 1$, we also increase i each time j increases, so that the sliding window is shifted to the right, by using sliding window algorithm for the suffix tree [52].

When $T[j + 1]$ is appended to the right end of $T[i..j]$, we first update the suffix tree to $\mathsf{STree}(T[i..j + 1])$ and compute $\mathsf{pp}_{i,j+1}$. Since $\mathsf{pp}_{i,j+1}$ coincides with the *active point*, $\mathsf{pp}_{i,j+1}$ can be found in amortized $O(\log \sigma')$ time [52].

After updating the suffix tree, we can compute $\mathsf{tp}_{i,j+1}$ and $\mathsf{sp}_{i,j+1}$ as follows:

- Traverse character $T[j + 1]$ from $\mathsf{tp}_{i,j}$, and set $w \leftarrow str(\mathsf{tp}_{i,j})T[i + 1]$.
- While $\#occ_{T[i..j+1]}(w) < 3$, set $w \leftarrow w[1..]$ and search for the locus p for w by using suffix links in $\mathsf{STree}(T[i..j + 1])$.
- After breaking from the while-loop, obtain $\mathsf{tp}_{i,j+1} = \mathsf{p}$.
- $\mathsf{sp}_{i,j+1}$ equals the locus stored in p at the penultimate iteration of the while-loop.

Let us show the correctness of the above algorithm. After the first step, w is the longest suffix which possibly corresponds to $\mathsf{tp}_{i,j+1}$. In the while loop of the second step, we search for the suffix corresponding to $\mathsf{tp}_{i,j+1}$ by deleting the first characters from w one-by-one. After breaking from the while-loop, we store in w the longest suffix of $T[i..j + 1]$ which occurs more than twice in $T[i..j + 1]$, i.e., $\mathsf{tp}_{i,j+1} = locus(w)$. Also, by the definitions of sp and tp, $\mathsf{sp}_{i,j+1}$ is the locus for the suffix of $T[i..j + 1]$ which is one character longer than $w = str(\mathsf{tp}_{i,j+1})$.

As is described in the above algorithm, we can locate $\mathsf{tp}_{i,j+1}$ using suffix links, in a similar manner to the active point $\mathsf{pp}_{i,j+1}$. Thus, the time cost for locating $\mathsf{tp}_{i,j+1}$ for each increasing j is amortized $O(\log \sigma')$, again by a similar argument to the active point $\mathsf{pp}_{i,j+1}$. What remains is, for each candidate w for $\mathsf{tp}_{i,j+1}$, how to quickly determine whether $\#occ_{T[i..j+1]}(w) < 3$ or not. In what follows, we show that it can be checked in $O(1)$ time for each candidate.

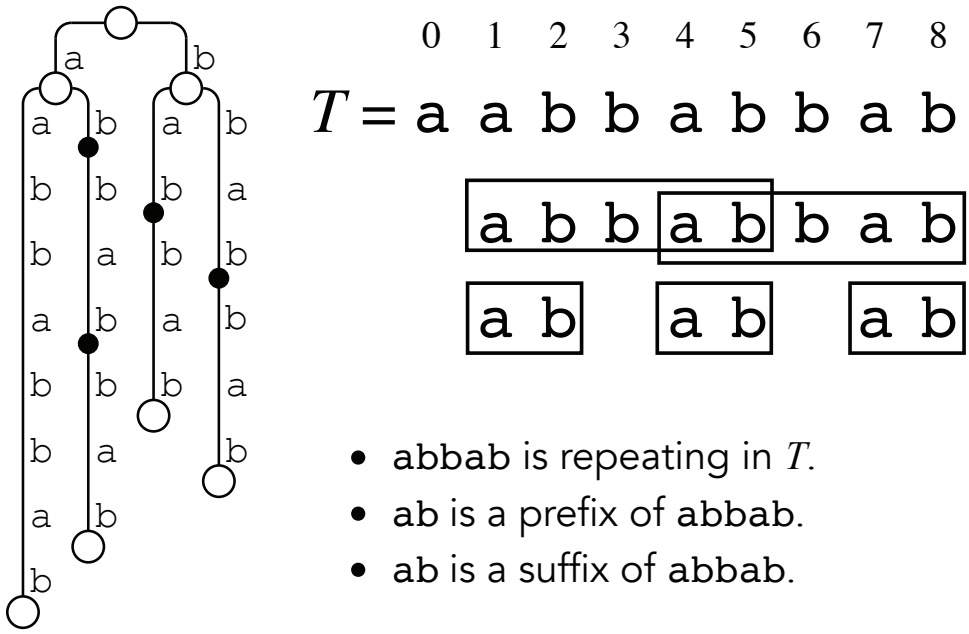

Figure 6.5: The suffix tree of string $T = \texttt{aabbabbab}$ as an example of the Case 3.1 in Observation 6.4. Black circles represent implicit suffix nodes. For two suffixes $s = \texttt{ab}$ and $s' = \texttt{abbab}$ of T, $hed(s') = hed(s)$ and s occurs three times in T.

Observation 6.4. *For each suffix s of string $T[i..j+1]$, let $locus(s) = \langle u, h\rangle$.*

Case 1. *If u is an internal node, s occurs at least three times in $T[i..j+1]$.*

Case 2. *If u is a leaf and $h = 0$, s occurs exactly once in $T[i..j+1]$.*

Case 3. *If u is a leaf and $h \neq 0$,*

> **Case 3.1.** *if there is a suffix s' of $T[i..j+1]$ with $hed(s') = hed(s)$ which is longer than s, s occurs at least three times in $T[i..j+1]$ (see Fig. 6.5 for examples).*
>
> **Case 3.2.** *otherwise, s occurs exactly twice in $T[i..j+1]$.*

For any suffix s of $T[i..j+1]$, if we are given $locus(s) = \langle u, h\rangle$, then we can obviously determine in constant time whether s occurs at least three times in $T[i..j+1]$ or not, except Case 3. The next lemma allows us to determine it in constant time in Case 3 as well.

Lemma 6.10. *Suppose the locus $\mathsf{pp}_{i,j+1}$ in $\mathsf{STree}(T[i..j+1])$ is already computed. Given a leaf ℓ of $\mathsf{STree}(T[i..j+1])$, it can be determined in $O(1)$ time whether there is an implicit suffix node on the edge $(parent(\ell), \ell)$ and if so, the locus of the lowest implicit suffix node on $(parent(\ell), \ell)$ can be computed in $O(1)$ time.*

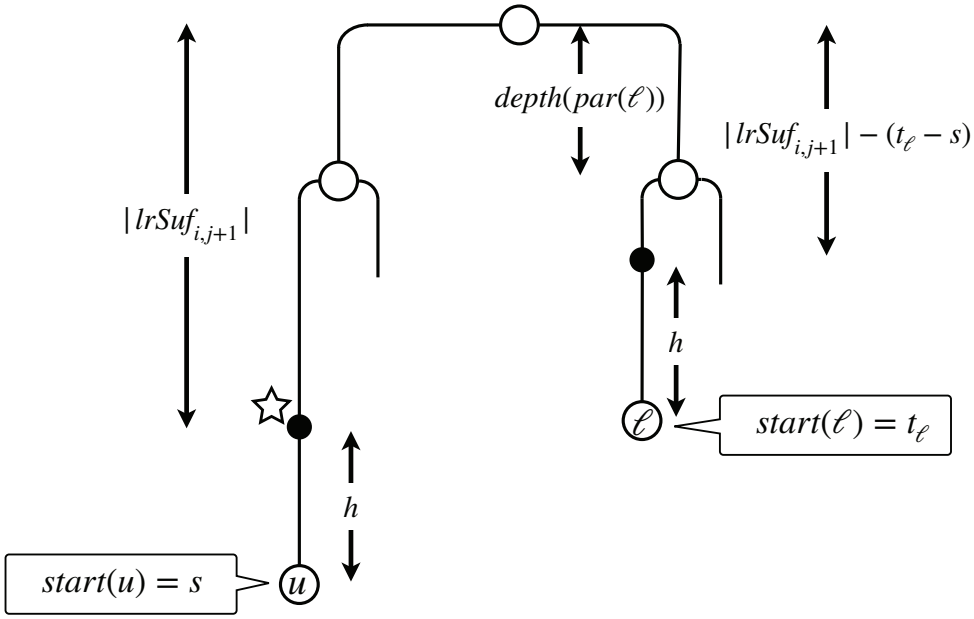

Figure 6.6: For an example of Lemma 6.10. The situation of this figure is that each of u and ℓ is a leaf with $t_\ell = start(\ell) \geq s = start(u)$ and $|lrSuf_{i,j+1}| - (t_\ell - s) > depth(parent(\ell))$ where $\langle u, h\rangle$ represents the primary active point. Also, black nodes represent implicit suffix nodes.

Proof. By Observation 6.4, for each leaf ℓ, the suffix corresponding to the lowest implicit suffix node on $(parent(\ell), \ell)$ occurs exactly twice in $T[i..j+1]$ if such an implicit suffix node exists. Let $x = lrSuf_{i,j+1}$ and $\mathsf{pp}_{i,j+1} = \langle u, h\rangle$.

If u is not a leaf, there is no implicit suffix node on the edge $(parent(\ell), \ell)$ for any leaf ℓ, since every suffix of $T[i..j+1]$ which is shorter than $|x|$ occurs more than twice in $T[i..j+1]$.

If u is a leaf, then $\#occ_{T[i..j+1]}(x) = 2$. Let $s = start(u)$ and $t_\ell = start(\ell)$ for each leaf ℓ. Notice that x is a border of $T[s..j+1]$. There are two sub-cases:

- First, we consider the case where $t_\ell < s$. Suppose that there is an implicit suffix node on $(parent(\ell), \ell)$ for the sake of contradiction. Let w be a string corresponding to the lowest implicit suffix node on $(parent(\ell), \ell)$. Then, w is a proper suffix of x, and occurs exactly twice in $T[i..j+1]$. Furthermore, w occurs exactly twice in $T[s..j+1]$ since x is a border of $T[s..j+1]$. However, w is also a prefix of $T[t_\ell..j+1]$, hence w occurs at least three times in $T[i..j+1]$, it is a contradiction. Thus, if $t_\ell < s$, there is no implicit suffix node on $(parent(\ell), \ell)$.

- Second, we consider the case where $t_\ell \geq s$ (see Fig. 6.6). In this case, $T[t_\ell..s+|x|-1]$ which is a prefix of $T[t_\ell..j+1]$ matches the suffix of x which is $t_\ell - s$ characters shorter than x, i.e., $x[t_\ell - s..]$. Thus, there is an implicit suffix node on $(parent(\ell), \ell)$ if and only

if $|T[t_\ell..s+|x|-1]| = |x| - (t_\ell - s) > depth(parent(\ell))$. Also, if there is an implicit suffix node on $(parent(\ell), \ell)$, the locus of the lowest one is $\langle \ell, h \rangle$.

□

Deleting the Leftmost Character

When the leftmost character $T[i-1]$ is deleted from $T[i-1..j]$, we first update the suffix tree and compute $\mathsf{pp}_{i,j}$ by using the sliding window algorithm for the suffix tree [52]. Each pair of position pointers for the edge-labels of the suffix tree can be maintained in amortized $O(1)$ time so that these pointers always refer to positions within the current sliding window, by a simple *batch update* technique (see [52] for details). After that, we compute $\mathsf{tp}_{i,j}$ and $\mathsf{sp}_{i,j}$ in a similar way to the case of appending a new character shown previously.

It follows from the above arguments in this subsection that we can update the suffix tree and the three active points in amortized $O(\log \sigma')$ time, each time the window is shifted by one character.

6.3.2 Computing $sqPref_{i-1,j}$

In order to compute the changes of MUSs when the leftmost character $T[i-1]$ is deleted from $T[i-1..j]$, we use $sqPref_{i-1,j}$ (cf. Lemma 6.7 and Lemma 6.9) before updating the suffix tree. In this subsection, we present an efficient algorithm for computing $sqPref_{i-1,j}$. First, we consider the following cases (see Fig. 6.7), where ℓ is the leaf corresponding to $T[i-1..j]$:

Case A. $hed(lrSuf_{i-1,j}) = \ell$.

Case B. $hed(lrSuf_{i-1,j}) \neq \ell$ and $subtree(parent(\ell))$ has more than two leaves.

Case C. $hed(lrSuf_{i-1,j}) \neq \ell$ and $subtree(parent(\ell))$ has exactly two leaves.

For Case A, the next lemma holds:

Lemma 6.11. *Given* $\mathsf{STree}(T[i-1..j])$ *and* $\mathsf{pp}_{i-1,j}$. *Let* ℓ *be the leaf corresponding to* $T[i-1..j]$. *If* $\mathsf{pp}_{i-1,j}$ *is on the edge* $(parent(\ell), \ell)$, *the following propositions hold:*

(a) $occ_{T[i-1..j]}(sqPref_{i-1,j}) = \{i-1, j-|lrSuf_{i-1,j}|+1\}$.

(b) *If there is exactly one implicit suffix node on* $(parent(\ell), \ell)$, $sqPref_{i-1,j} = T[i-1..i-1+depth(parent(\ell))]$.

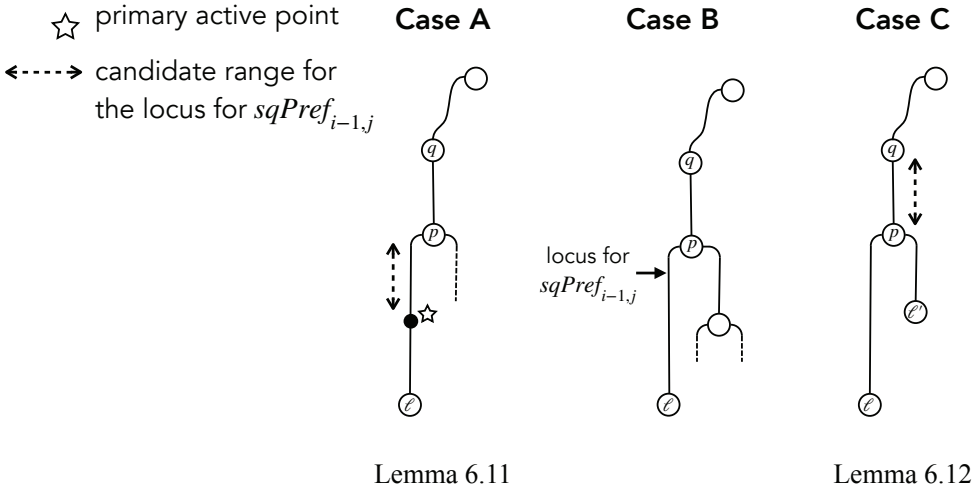

Figure 6.7: Illustration for the three cases that are described in Section 6.3.2.

(c) *If there are more than one implicit suffix node on* $(parent(\ell), \ell)$*, then* $|lrSuf_{i-1,j}| > \lfloor (j - i + 2)/2 \rfloor$ *and* $sqPref_{i-1,j} = T[i-1..j-2h+1]$*, where* $\mathsf{pp}_{i-1,j} = \langle \ell, h \rangle$.

Proof. Let $\mathsf{pp}_{i-1,j} = \langle \ell, h \rangle$ and $m = |lrSuf_{i-1,j}|$.

(a) Since $\mathsf{pp}_{i-1,j}$ is on the edge $(parent(\ell), \ell)$, $sqPref_{i-1,j}$ is a prefix of $lrSuf_{i-1,j}$, and $\#occ_{T[i-1..j]}(lrSuf_{i-1,j}) = \#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 2$. Therefore, we obtain that $occ_{T[i-1..j]}(sqPref_{i-1,j}) = occ_{T[i-1..j]}(lrSuf_{i-1,j}) = \{i-1, j-m+1\}$.

(b) In this case, it is clear that $sqPref_{i-1,j} = T[i-1..i-1+depth(parent(\ell))]$.

(c) Let $\langle \ell, h' \rangle$ be the locus of the implicit suffix node which is the lowest on the edge $(parent(\ell), \ell)$ except $\mathsf{pp}_{i-1,j}$. Also, let x be the string corresponding to the locus $\langle \ell, h' \rangle$. In this case, x occurs exactly three times in $T[i-1..j]$. Also, x is the longest border of $lrSuf_{i-1,j}$. Assume on the contrary that $m \leq \lfloor (j-i+2)/2 \rfloor$. Then, two occurrences of $lrSuf_{i-1,j}$ in $T[i-1..j]$ are not overlapping, and thus $\#occ_{T[i-1..j]}(x) \geq 2 \times \#occ_{T[i-1..j]}(lrSuf_{i-1,j}) = 4$, it is a contradiction. Therefore, $m > \lfloor (j-i+2)/2 \rfloor$ (see Fig. 6.8).

Next, we consider a relation between h and h'. By the definition, $h = |T[i-1..j]| - m = j - i + 2 - m$. Since $m > \lfloor (j-i+2)/2 \rfloor$, x matches the intersection of two occurrences of $lrSuf_{i-1,j}$, i.e., $x = T[j-m+1..i+m-2]$. Thus, $h' = |T[i-1..j]| - |x| = j - i + 2 - (2m - j + i - 2) = 2(j - i + 2 - m) = 2h$. Therefore $sqPref_{i-1,j} = T[i-1..j-h'+1] = T[i-1..j-2h+1]$.

□

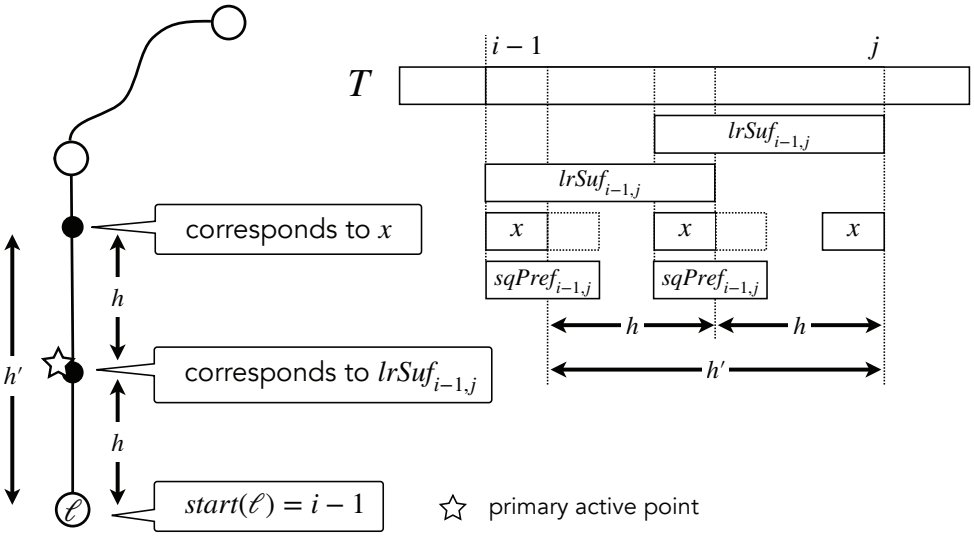

Figure 6.8: Illustration for the proposition (c) in Lemma 6.11. For the sake of simplicity, this figure shows a simple case where there are only two implicit suffix nodes on the edge $(parent(\ell), \ell)$. However, the lemma also holds for the other cases.

In Case B, it is clear that $sqPref_{i-1,j} = T[i-1..i-1+depth(p)]$ since $str(p)$ occurs at least three times in $T[i-1..j]$ (see Fig. 6.7).

For Case C, the next lemma holds:

Lemma 6.12. *Suppose that* $\mathsf{STree}(T[i-1..j])$ *and* $\mathsf{pp}_{i-1,j}$ *have already been computed. Let* ℓ *be the leaf corresponding to* $T[i-1..j]$*,* $p = parent(\ell)$*, and* $q = parent(p)$*. If* $subtree(p)$ *has exactly two leaves and there are no implicit suffix nodes on any edges in* $subtree(p)$*, then it can be determined in* $O(1)$ *time whether there is an implicit suffix node on* (q, p)*. If such an implicit node exists, then the locus of the lowest implicit suffix node on* (q, p) *can be computed in* $O(1)$ *time.*

Proof. Note that the suffix corresponding to the lowest implicit suffix node on (q, p) occurs exactly three times in $T[i-1..j]$ from the assumptions. Let $\mathsf{pp}_{i-1,j} = \langle u, h\rangle$. If $h = 0$, the primary active point is an explicit node, and there is no implicit suffix node on every edge in $\mathsf{STree}(T[i-1..j])$. If $h \neq 0$ and $u = p$, the lowest implicit suffix node on (q, p) is clearly the primary active point. Thus, in the following, we consider the situation with $u \neq p$ and $h \neq 0$.

If u is not a leaf and the number of leaves in $subtree(u)$ is greater than two, then the number of leaves in $subtree(hed(v))$ is also greater than two for each implicit suffix node v. Thus, there is no implicit suffix node on (q, p). If u is not a leaf and the number of leaves in $subtree(u)$ is exactly two, then $lrSuf_{i-1,j}$ occurs at least three times in $T[i-1..j]$ since $u \neq p$. Thus,

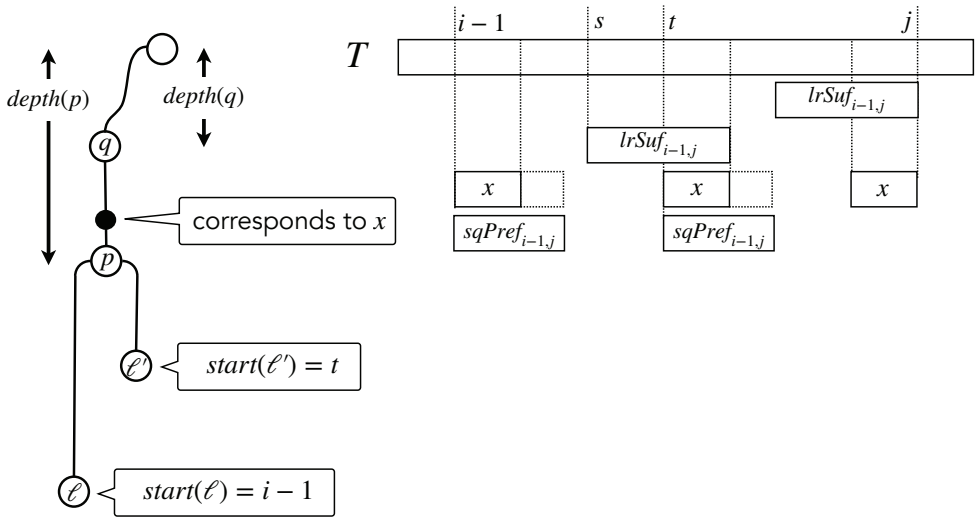

Figure 6.9: Illustration for Lemma 6.12.

if a suffix s of $T[i-1..j]$ which is shorter than $lrSuf_{i-1,j}$ occurs as a prefix of $T[i-1..j]$, $\#occ_{T[i-1..j]}(s) \geq 4$. Therefore, there is no implicit suffix node on (q, p).

If u is a leaf, as in the proof in Lemma 6.10, it can be proven that there is an implicit suffix node on (q, p) if and only if $t \geq s$ and $depth(p) > |lrSuf_{i-1,j}| - (t - s) > depth(q)$, where $s = start(u)$, $t = start(\ell')$ with ℓ' being the sibling of ℓ (see Fig. 6.9).

In addition, if there is an implicit suffix node on the edge (q, p), the length of the string x corresponding to the lowest implicit suffix node on the edge (q, p) is $|lrSuf_{i-1,j}| - (t - s)$, and thus, the implicit suffix node is $\langle p, depth(p) - |x| \rangle = \langle p, depth(p) - |lrSuf_{i-1,j}| + t - s \rangle$. $\square$

We can design an algorithm for computing $sqPref_{i-1,j}$ by using the above lemmas, as follows. Let ℓ be the leaf corresponding to $T[i-1..j]$, $p = parent(\ell)$ and $q = parent(p)$.

In Case A. $sqPref_{i-1,j}$ is computed by Lemma 6.11.

In Case B. $sqPref_{i-1,j} = T[i-1..i-1+depth(p)]$ and $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 1$.

In Case C. We divide this case into some subcases by the existence of an implicit suffix node on edges (p, ℓ') and (q, p) where ℓ' is the sibling of ℓ. We first determine the existence of an implicit suffix node on (p, ℓ') (by Lemma 6.10).

- If there is an implicit suffix node on (p, ℓ'), then $sqPref_{i-1,j} = T[i-1..i-1+depth(p)]$ and $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 1$.
- If there is no implicit suffix node on both (p, ℓ) and (p, ℓ'), we can determine in constant time the existence of an implicit suffix node on (q, p) (by Lemma 6.12). If

there is an implicit suffix node on (q, p), $sqPref_{i-1,j} = T[i-1..depth(p)-h+1]$ and $occ_{T[i-1..j]}(sqPref_{i-1,j}) = \{i-1, start(\ell')\}$. Otherwise, $sqPref_{i-1,j} = T[i-1..depth(q)+1]$ and $occ_{T[i.-1.j]}(sqPref_{i-1,j}) = \{i-1, start(\ell')\}$.

It follows from the above arguments in this subsection that $sqPref_{i-1,j}$ can be computed in $O(1)$ time by using the suffix tree and the (primary) active point.

6.3.3 Detecting MUSs to be Added/Deleted

By using the afore-mentioned lemmas in this section, we can design an efficient algorithm for detecting MUSs to be added / deleted.

Data Structure for Maintaining MUSs

First, we introduce a data structure for managing the set of MUSs for a sliding window. Our data structure for MUSs consists of two arrays S2E and E2S of length d each. Note that by the definition of MUSs, any MUSs cannot be nested each other. Thus, for any text position i, if a MUS starting (resp. ending) at i exists, then its ending (resp. starting) position is unique. From this fact, we can define S2E and E2S as follows:

Let $[p, p+d-1]$ be the current window. For every index i with $p \leq i \leq p+d-1$,

$$\mathsf{S2E}[i \bmod d] = \begin{cases} e & \text{if } [i, e] \in \mathsf{MUS}_{T[p..p+d-1]} \text{ exists,} \\ nil & \text{otherwise.} \end{cases}$$

$$\mathsf{E2S}[i \bmod d] = \begin{cases} s & \text{if } [s, i] \in \mathsf{MUS}_{T[p..p+d-1]} \text{ exists,} \\ nil & \text{otherwise.} \end{cases}$$

Since MUSs cannot be nested each other, these arrays are uniquely defined (see Fig. 6.10). By using these two arrays, all the following operations for MUSs can be executed in $O(1)$ time; add/remove a MUS into/from the set of MUSs, and compute the ending/starting position of the MUS that starts/ends at a specified position.

Algorithm When Appending a Character to the Right

Assume that S2E, E2S and the suffix tree of $T[i..j]$ are computed before reading $\gamma = T[j+1]$. Also, assume that the longest single character run β^e as a suffix of $T[i..j]$ is known, where $\beta = T[j]$ and $e \geq 1$.

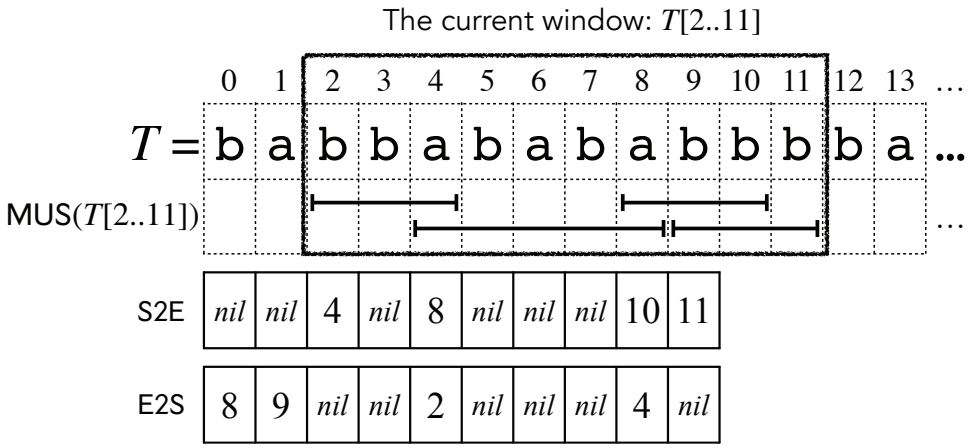

Figure 6.10: A long string $T = \texttt{babbababbbba}\cdots$ and two arrays S2E and E2S. The current window is $T[2..11]$ of length $d = 10$, and the MUSs in the window are $T[2..4], T[4..8], T[8..10]$, and $T[9..11]$.

- First, compute the length of $lrSuf_{i,j}$.
- Second, read γ, and update the suffix tree and the active points. Then, compute the lengths of $lrSuf_{i,j+1}$ and $sqSuf_{i,j+1}$. Also, update information about the run of the last character of $T[i..j+1]$. Specifically, if $\gamma = \beta$ then $\beta^e = \gamma^{e+1}$, and otherwise $\beta^e = \gamma^1$. If $|lrSuf_{i,j+1}| \leq |lrSuf_{i,j}|$ or $T[j+1-|lrSuf_{i,j+1}|..j+1] = \gamma^{|lrSuf_{i,j+1}|+1}$, add $[j+1-|lrSuf_{i,j+1}|, j+1]$ into the set of MUSs (by Lemma 6.4).
- If $|lrSuf_{i,j+1}| < |sqSuf_{i,j+1}|$, then terminate this step (by Lemma 6.5).
- Otherwise, compute p_s and q of Lemma 6.6 by using $\mathsf{STree}(T[i..j+1])$ and $\mathsf{sp}_{i,j+1}$. Then, remove $[p_s, q]$ from the set of MUSs (by Lemma 6.3).
- Next, if $\mathsf{E2S}[t' \bmod d] = nil$, then add $[p_s, t']$ into the set of MUSs, where $t' = q+1$. Also, if $s' \geq i$ and $\mathsf{S2E}[s' \bmod d] = nil$, then add $[s', q]$ into the set of MUSs, where $s' = q - |lrSuf_{i,j+1}|$ (by Lemma 6.6).
- Terminate this step.

Algorithm When Deleting the Leftmost Character

Assume that S2E, E2S and the suffix tree of $T[i-1..j]$ are computed before deleting $\alpha = T[i-1]$.

- First, compute $\#occ_{T[i-1..j]}(sqPref_{i-1,j})$. If $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 2$, compute two integers s and t with $T[s..t] = sqPref_{i-1,j}$ and $s \neq i-1$.
- Second, delete $T[i-1]$ and update the suffix tree and the active points. If $\mathsf{S2E}[(i-1) \bmod d] \neq nil$, remove the MUS starting at $i-1$ from the set of MUSs.
- If $\#occ_{T[i-1..j]}(sqPref_{i-1,j}) = 1$, terminate this step (by Lemma 6.8).
- Otherwise, if $\mathsf{S2E}[s \bmod d] \neq nil$, then remove the MUS starting at s from the set of MUSs. Also, if $\mathsf{E2S}[t \bmod d] \neq nil$, then remove the MUS ending at t from the set of MUSs (by Lemma 6.9).
- Finally, add $[s, t]$ into the set of MUSs (by Lemma 6.7), and terminate this step.

The main result of this section is the following:

Theorem 6.3. *We can maintain the set of MUSs for a sliding window of length d on a string T of length n in a total of $O(n \log \sigma')$ time and $O(d)$ working space where $\sigma' \leq d$ is the maximum number of distinct characters in every window.*

Corollary 6.2. *There exists an* online *algorithm to compute all MUSs in a string T of length n in a total of $O(n \log \sigma)$ time with $O(n)$ working space where σ is the alphabet size.*

6.4 Conclusions and Future Work

In this chapter, we studied the problem of computing MUSs for a sliding window over a given string T of length n. We first showed combinatorial properties on MUSs for a sliding window, i.e., changes of the set of MUSs are at most constant when appending a character to the right end of the window or deleting the first character from the window. Also, we proposed an $O(n \log \sigma')$-time and $O(d)$-space algorithm to compute MUSs for a sliding window of size d over T, where $\sigma' \leq d$ is the maximum number of distinct characters in every window.

As future work, we are interested in developing a data structure for the SUS problems for a sliding window. As we described in the introduction, MUSs are heavily utilized for solving the SUS problems. Our sliding window MUS algorithm could be used as a basis for an efficient SUS query data structure for a sliding window. Also, it would be interesting to extend or generalize MUSs for a sliding window, e.g., to computing MUSs with k-*mismatches* for a sliding window. A substring of T is said to be unique with k-mismatches in T, if it is unique in T even

when substituting arbitrary k characters of the substring. To the best of our knowledge, only one *deterministic* algorithm to compute unique substrings with k-mismatches is known in [26], and their algorithm runs in $O(n^2)$ time for any $k \geq 1$ in an offline manner. An interesting open question is: Can we design an online deterministic algorithm which computes MUSs with k-mismatches in sub-quadratic time?.

Chapter 7

Computing Minimal Unique Palindromic Substrings for a Semi-Dynamic String via Palindromic Tree

A *palindrome* is a string that reads the same forward and backward. Palindromic structures in strings have been heavily studied in the fields of string processing algorithms and combinatorics on strings [42, 24, 36, 50, 18, 5]. One of the most famous results related to palindromic structures is Manacher's algorithm [42], which computes all maximal palindromes in a given string T. Manacher's algorithm essentially computes all palindromes in T, since any palindromic substring of T is a substring of some maximal palindrome in T. Another interesting topic is enumeration of distinct palindromes in a string. It is known that any string of length n contains at most $n + 1$ distinct palindromes including the empty string [16]. Groult et al. [24] proposed an $O(n)$-time algorithm which enumerates all distinct palindromes in a given string of length n over an integer alphabet of size $\sigma = n^{O(1)}$. For the same problem in the *online* model, Kosolobov et al. [36] proposed an $O(n \log \sigma)$-time and $O(n)$-space algorithm for a general ordered alphabet. Kosolobov et al.'s algorithm is a combination of Manacher's algorithm and Ukkonen's online suffix tree construction algorithm [56]. Rubinchik and Shur [50] proposed a new data structure called *eertree*, which permits efficient access to distinct palindromes in a string *without* storing the string itself. Eertrees can be utilized for solving problems related to palindromic structures, such as the palindrome counting problem and the palindromic factorization problem [50]. The size of the eertree of T is linear in the number p_T of distinct palindromes in T [50]. Since p_T is at most $|T| + 1$, the size of the eertree of T can be much smaller than the length $|T|$ of the string, e.g., for $T = (\texttt{abc})^{n/3}$, $p_T = 4$ since all distinct palindromes in T are

a, b, c, and the empty string. Thus, the size of the eertree of T can be much smaller than that of the suffix tree of T which is $\Theta(n)$. Therefore, it is of significance if one can build eertrees without suffix trees. Rubinchik and Shur [50] indeed proposed an online eertree construction algorithm running in $O(n \log \sigma)$ time without suffix trees.

Recently, a concept of palindromic structures called *minimal unique palindromic substrings* (*MUPS*) is introduced by Inoue et al. [30]. A palindromic substring $w = T[i..j]$ of a string T is called a MUPS of T if w occurs in T exactly once, and $T[i+1..j-1]$ occurs at least twice in T. MUPSs are utilized for solving the *shortest unique palindromic substring* (*SUPS*) problem [30], which is motivated by an application in molecular biology. Watanabe et al. [57] proposed an algorithm to solve the SUPS problem based on the *run-length encoding* (*RLE*) version of eertrees, named $\mathtt{e}^2\mathtt{rtre}^2$.

In this chapter, we consider the problem of computing MUPSs in a semi-dynamic string. As in Chapter 6, we will identify a problem in semi-dynamic strings with a problem in the sliding window model. Namely, we tackle the following problem: For each window $T[0..d-1], \ldots, T[n-d..n-1]$ of length d over a string T, the aim is to maintain the set of MUPSs in the window.

For the sake of computing MUPSs efficiently, we first consider the problem of maintaining the eertrees in the sliding window model. We propose an algorithm which maintains eertrees for a sliding window in a total of $O(n \log \sigma')$ time using $O(d)$ space. Also, we give an alternative eertree construction algorithm for a sliding window which runs in $O(n + d\sigma)$ time with $(d+2)\sigma + O(d)$ space.

Furthermore, by utilizing the result for eertrees, we propose an algorithm which maintains MUPSs for a sliding window in a total of $O(n \log \sigma')$ time using $O(d)$ space. In addition, we introduce a new concept of palindromic structures called *minimal absent palindromic words* (*MAPW*), and consider the problem of maintaining MAPWs for a sliding window. A string w is called a MAPW of string T if w is a palindrome, w does not occur in T, and $w[1..|w|-2]$ occurs in T. MAPWs can be seen as a palindromic version of the notion of minimal absent words (MAWs), which are extensively studied in the fields of string processing and bioinformatics [14, 44, 9, 48, 20]. We propose an algorithm which maintains MAPWs for a sliding window in a total of $O(n + d\sigma)$ time using $O(d\sigma)$ space.

7.1 Preliminaries

Palindromes. A string T is called a *palindrome* if $T[i] = T[|T| - i - 1]$ for every $0 \leq i \leq |T| - 1$. Note that the empty string is a palindrome. A substring $T[i..j]$ of T is said to be a *palindromic substring* of T if $T[i..j]$ is a palindrome. The *center* of a palindromic substring $T[i..j]$ of T is $\frac{i+j}{2}$. A palindromic substring $T[i..j]$ of T is *maximal* w.r.t. the center position $\frac{i+j}{2}$ if $i = 0$, $j = |T| - 1$, or $T[i-1..j+1]$ is not a palindrome. We denote by $lpp(T)$ (resp. $lps(T)$) the longest palindromic prefix (resp. suffix) of T. We denote by $\mathsf{DPal}(T)$ the set of all distinct palindromes in T. It is known that $|\mathsf{DPal}(T)| \leq |T| + 1$ [16].

A substring $T[i..j]$ of T is called a *minimal unique palindromic substring (MUPS)* of T if $T[i..j]$ is a palindrome, $T[i..j]$ is unique in T, and $T[i+1..j-1]$ is repeating in T. We denote MUPS_T the set of intervals corresponding to MUPSs of T, i.e., $\mathsf{MUPS}_T = \{[i,j] \mid T[i..j] \text{ is a MUPS of } T\}$. For example, palindromic substring $T[9..13] = \texttt{bbabb}$ of string $T = \texttt{aaabababababbabb}$ is a MUPS of T since $T[9..13] = \texttt{bbabb}$ is unique in T and $T[10..12] = \texttt{bab}$ is repeating in T.

A string w is called a *minimal absent palindromic word* (*MAPW*) of string T if w is a palindrome, w does not occur in T, and $w[1..|w|-2]$ occurs in T. For example, palindrome $w = \texttt{aabbaa}$ is a MAPW of string $T = \texttt{aaabababababbabb}$ since w does not occur in T and $w[1..|w|-2] = \texttt{abba}$ occurs in T at position 8.

Eertree (Palindromic Tree). The *eertree* of T denoted by $\mathsf{eertree}(T)$ is a tree-like data structure that enables us to efficiently access each of the distinct palindromes in T [50]. The $\mathsf{eertree}(T)$ consists of m *ordinary nodes* and two auxiliary nodes, denoted $\texttt{0}$-*node* and $-\texttt{1}$-*node*, where $m = |\mathsf{DPal}(T)| - 1$. Each ordinary node corresponds to each element of $\mathsf{DPal}(T) \setminus \{\varepsilon\}$. For each ordinary node v, we denote by $pal(v)$ the palindrome which corresponds to v, and by $len(v)$ its length. Conversely, for each non-empty palindromic substring p of T, we denote by $node(p)$ the node which corresponds to the palindrome p. Namely, $node(pal(v)) = v$ for each ordinary node v. For convenience, we define $pal(\texttt{0}\text{-node}) = pal(-\texttt{1}\text{-node}) = \varepsilon$, $len(\texttt{0}\text{-node}) = 0$, and $len(-\texttt{1}\text{-node}) = -1$. For any nodes u, v in $\mathsf{eertree}(T)$, there is an edge (u, v) if and only if $len(u) + 2 = len(v)$ and $pal(u) = pal(v)[1..len(v) - 2]$. Each edge (u, v) is labeled by a character $pal(v)[0]$. Also, each node v in $\mathsf{eertree}(T)$ has a *suffix link* denoted by $slink(v)$. For each node v in $\mathsf{eertree}(T)$ with $len(v) \geq 2$, $slink(v)$ points to the node corresponding to the longest palindromic proper suffix of $pal(v)$. For each node v in $\mathsf{eertree}(T)$ with $len(v) = 1$, $slink(v)$ points to the $\texttt{0}$-node. Also, $slink(\texttt{0}\text{-node}) = -\texttt{1}$-node and

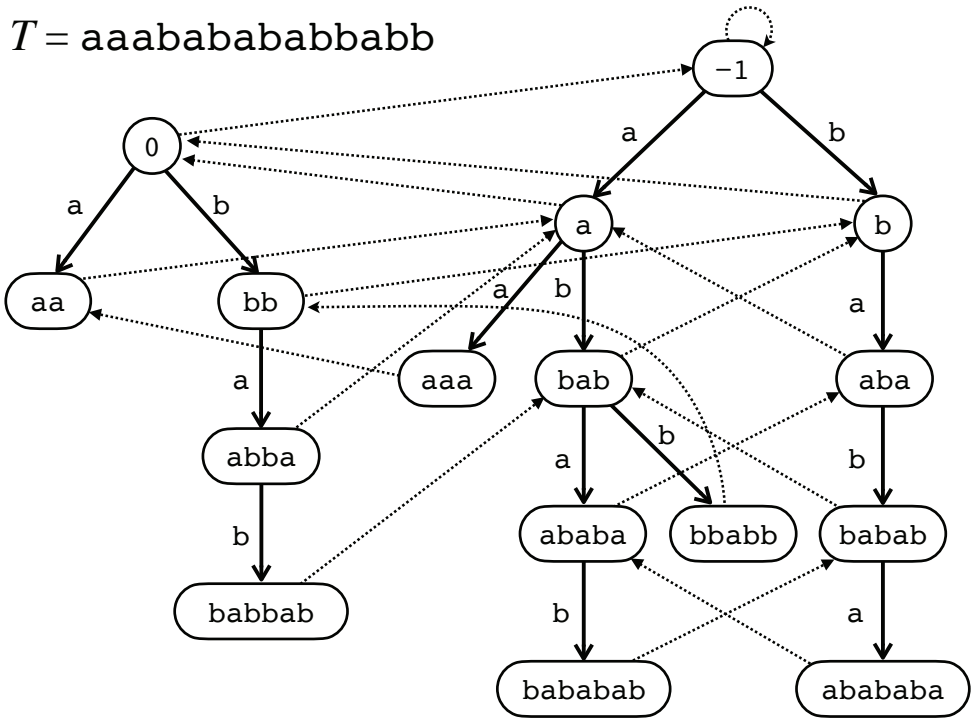

Figure 7.1: The eertree of T = aaababababbabb. The solid and broken arrows represent edges and suffix links, respectively. Note that $pal(v)$ is written inside each node v in this figure, however, it is for only explanation. Namely, each node does not explicitly store the corresponding string.

$slink(-1\text{-node}) = -1\text{-node}$. For each node v in $\mathsf{eertree}(T)$, $inSL(v) = |\{u \mid slink(u) = v\}|$ denotes the number of incoming suffix links of v. See Fig. 7.1 for an example of $\mathsf{eertree}(T)$.

Note that each node v does not store the string $pal(v)$ explicitly. Instead, we can obtain $pal(v)$ by traversing edges backward, from v to the root, since $pal(u) = c\,pal(u')c$ for each node u with $|\,pal(u)| \geq 2$ where u' is the parent of u and c is the label of the edge (u', u). Each node only stores pointers to its children and a constant number of integers. Thus, the size of $\mathsf{eertree}(T)$ is linear in the number of nodes, i.e., $O(|\mathsf{DPal}(T)|)$. It is known that $\mathsf{eertree}(T)$ can be constructed in $O(n \log \sigma)$ time for any string T given in an online manner [50].

7.2 Combinatorial Properties on Palindromes for a Sliding Window

In this section, we show some combinatorial properties on palindromes for a sliding window, which is helpful for designing efficient algorithms to maintain eertrees for a sliding window.

Since the nodes of the eertree of a string represent all distinct palindromes in the string, we obtain the next lemma.

Lemma 7.1. *There is a node ℓ in* $\mathsf{eertree}(T[i-1..j-1])$ *to be removed when the leftmost character $T[i-1]$ is deleted from $T[i-1..j-1]$ if and only if (A) $pal(\ell)$ is unique in $T[i-1..j-1]$ and (B) $pal(\ell) = lpp(T[i-1..j-1])$. Moreover when this holds, (C) ℓ is a leaf node.*

Proof. ($\Rightarrow$) (A) Since ℓ is removed, $pal(\ell)$ does not occur in $T[i..j-1]$. Thus, $pal(\ell)$ occurs in $T[i-1..j-1]$ only as a prefix, i.e., $pal(\ell)$ is unique in $T[i-1..j-1]$. (B) Assume that $pal(\ell)$ is shorter than $lpp(T[i-1..j-1])$. Then, $pal(\ell)$ is a proper prefix of $lpp(T[i-1..j-1])$. Also, $pal(\ell)$ is a proper suffix of $lpp(T[i-1..j-1])$ since $lpp(T[i-1..j-1])$ is a palindrome. This contradicts that $pal(\ell)$ is unique in $T[i-1..j-1]$. Thus, $pal(\ell) = lpp(T[i-1..j-1])$. (C) If we assume that ℓ has a child, then $pal(\ell)$ has an occurrence in $T[i-1..j-1]$ that is not a prefix of $T[i-1..j-1]$, a contradiction. ($\Leftarrow$) Since $pal(\ell)$ is a palindromic prefix of $T[i-1..j-1]$ and unique in $T[i-1..j-1]$, $pal(\ell)$ does not occur in $T[i..j-1]$. Thus, ℓ is removed when $T[i-1]$ is deleted. □

Namely, when the leftmost character of the window is deleted, at most one leaf will be removed from the eertree. For example, in Fig. 7.1, the leaf corresponding to `aaa` will be removed when the leftmost character of T is deleted, since `aaa` is the longest palindromic prefix of T and it is unique in T. Also, in order to detect such a leaf, we need to compute the longest palindromic prefix of each window and to determine its uniqueness. In the following, we show some combinatorial properties on unique palindromes and the longest palindromic prefix for a sliding window.

Unique Palindromes for a Sliding Window. A palindromic substring w of string T is said to be *border-maximal* in T if there is no palindromic substring of T, which contains w as a proper suffix. See Fig. 7.2 for examples. If a palindrome w is not border-maximal in T, then w is a border of another palindrome w', i.e., w is not unique in T. In other words, any unique palindromic substring must be border-maximal.

Longest Prefix Palindrome for a Sliding Window. Next, we consider the longest palindromic prefixes for sliding windows.

Lemma 7.2. *Let w be the longest palindromic prefix of the window $T[i_t..j_t]$ at time t. There exists time $t' \leq t$ which satisfies one of the followings:*

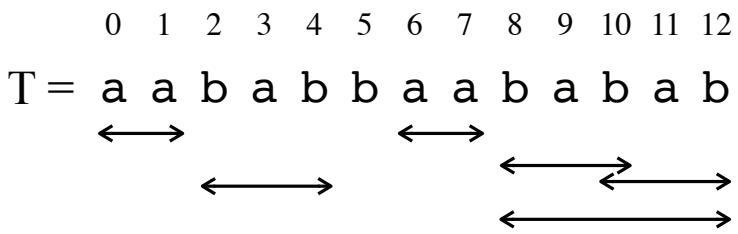

Figure 7.2: For string $T = \texttt{aababbaababab}$, its palindromic substring aa is border-maximal in T. On the other hand, bab is not border-maximal in T since there is a palindromic substring $T[8..12] = \texttt{babab}$ of T which contains bab as a proper suffix.

1. *the longest palindromic suffix of* $T[i_{t'}..j_{t'}]$ *is* w*, or*

2. *the longest palindromic suffix of* $lpp(T[i_{t'}..j_{t'}])$ *is* w*.*

(See also Fig. 7.3 for concrete examples.)

Proof. Let $w = T[s..e]$. If $w = T[i_t..j_t]$, then w is also the longest palindromic suffix of the window. Namely, w satisfies the first condition of the lemma for $t' = t$. Otherwise, for the sake of contradiction, we assume the contrary. Namely, for every time $t' \leq t$, neither the longest palindromic suffix of $T[i_{t'}..j_{t'}]$ nor the longest palindromic suffix of $lpp(T[i_{t'}..j_{t'}])$ is not equal to $T[s..e]$. Consider a window in the past such that its ending position is e. Since the longest palindromic suffix of the window is not $T[s..e]$, there is another palindromic suffix ending at e, which is longer than $T[s..e]$. Now let $v = T[s'..e]$ be the shortest palindrome which is ending at e and is longer than $w = T[s..e]$. Namely, w is the longest palindromic suffix of v. Next, consider a window $T[i_{\tilde{t}}..j_{\tilde{t}}]$ at time $\tilde{t} \leq t$ with its starting position $i_{\tilde{t}} = s'$. If v is the longest palindromic prefix of the window $T[i_{\tilde{t}}..j_{\tilde{t}}]$, then w becomes the longest palindromic suffix of $v = lpp(T[i_{\tilde{t}}..j_{\tilde{t}}])$, however, it contradicts our assumption. Thus, there is another palindromic prefix starting at s', which is longer than v. Now let $u = T[s'..e']$ be the longest palindromic substring of the window $T[i_{\tilde{t}}..j_{\tilde{t}}]$.

Further let c_w, c_v, and c_u be respectively the center of w, v, and u. From the assumptions, $c_v < c_w$ and $c_v < c_u$ hold. Next, we consider three sub-cases (see also Fig. 7.4):

(a) If $c_u < c_w$, then the palindrome u' ending at e whose center equals c_u, is longer than w and is shorter than v. This contradicts that w is the longest palindromic suffix of v.

(b) If $c_u = c_w$, then $|v| = e - s' + 1 = e' - s + 1$. Thus, $T[s..e'] = T[s'..e] = v$ since $T[s'..e']$ is a palindrome. This contradicts that w is the longest palindromic prefix of the current window $T[i_t..j_t] = T[s..j_t]$.

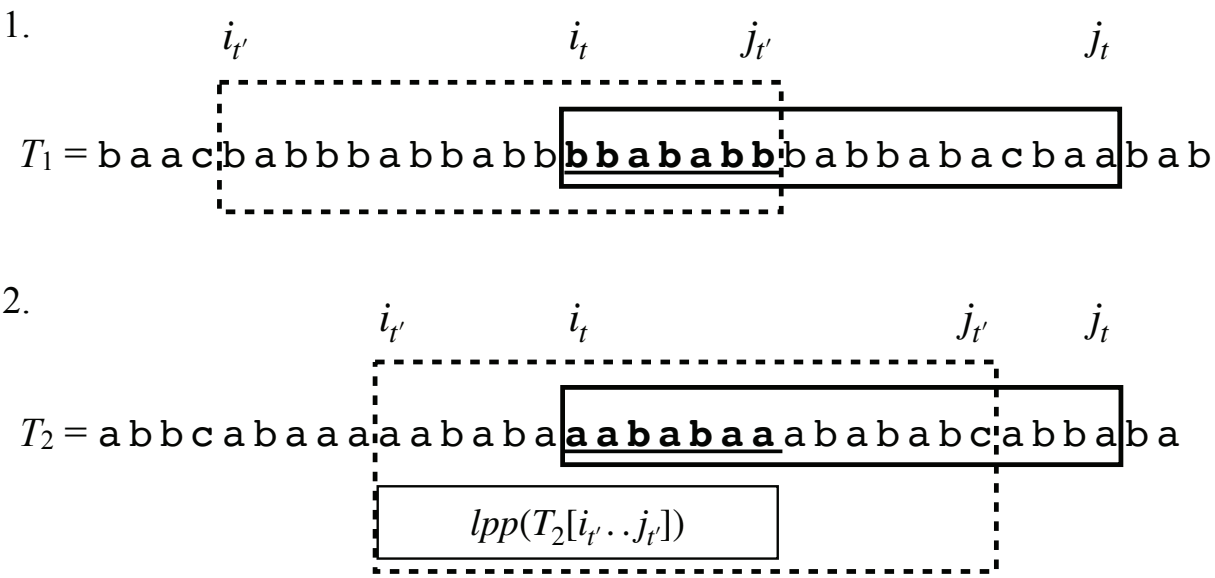

Figure 7.3: The upper one shows a string T_1 and windows $T_1[i_t..j_t]$ and $T_1[i_{t'}..j_{t'}]$ with $t' \leq t$. The longest palindromic prefix `bbababb` of $T_1[i_t..j_t]$ is equal to the longest palindromic suffix of $T_1[i_{t'}..j_{t'}]$ (i.e., the first case of Lemma 7.2). The lower one shows a string T_2 and windows $T_2[i_t..j_t]$ and $T_2[i_{t'}..j_{t'}]$ with $t' \leq t$. The longest palindromic prefix `aababaa` of $T_2[i_t..j_t]$ is equal to the longest palindromic suffix of $lpp(T_2[i_{t'}..j_{t'}])$ (i.e., the second case of Lemma 7.2).

(c) If $c_w < c_u$, then the palindrome u'' starting at s whose center equals c_u, is longer than w. This again contradicts that w is the longest palindromic prefix of the current window $T[i_t..j_t] = T[s..j_t]$.

Therefore, we have proved the lemma. □

7.3 Eertree for a Sliding Window

In this section, we show how to update a given eertree when we shift the sliding window to the right by one character. Sliding a given window consists of two operations: *deleting* the leftmost character and *appending* a character to the right end. Namely, when the eertree of $T[i-1..j-1]$ is given, we first compute the eertree of $T[i..j-1]$ (deleting the leftmost character), and then, compute the eertree of $T[i..j]$ (appending a character). To update the eertree when a character is appended, we can apply Rubinchik and Shur's algorithm [50] which constructs the eertree of a given string in an online manner. In this section, we propose new additional data structures and algorithms which update the eertree when the leftmost character is deleted.

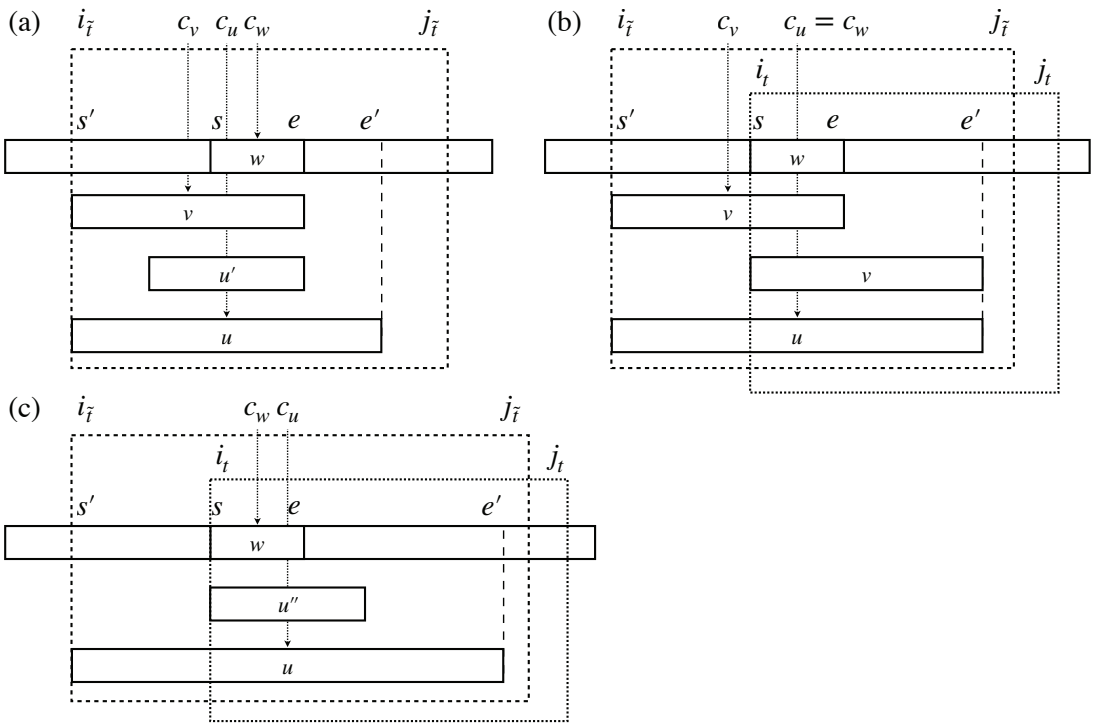

Figure 7.4: Illustration for contradictions in the proof of Lemma 7.2.

As in Chapter 6, our algorithms also work for a window of variable-length.

7.3.1 Auxiliary Data Structures for Detecting the Node to be Deleted

We introduce auxiliary data structures for computing the longest palindromic prefixes and for determining the uniqueness of palindromes.

For Computing the Longest Palindromic Prefix. Let $prefPal[0..d-1]$ be a cyclic array of size d such that $prefPal[i_t \bmod d]$ stores the node which corresponds to the longest palindromic prefix of the window $T[i_t..j_t]$ at each time t. Namely, for every time t, $prefPal[i_t \bmod d] = node(lpp(T[i_t..j_t]))$ holds.

For Determining Uniqueness of a Palindrome. For each ordinary node v in eertree$(T[i..j])$, let $rm_{i,j}(v)$ be the starting position of the rightmost occurrence of $pal(v)$ in $T[i..j]$. Further let $srm_{i,j}(v)$ be the starting position of the second rightmost occurrence of $pal(v)$ in $T[i..j]$ if such a position exists, and otherwise, $srm_{i,j}(v) = -1$. Throughout the computation of the eertree for a sliding window, for each node v of eertree$(T[i..j])$ we keep the following invariant $BegPair_{i,j}(v)$ which consists of two fields *first* and *second* such that: $BegPair_{i,j}(v).first =$

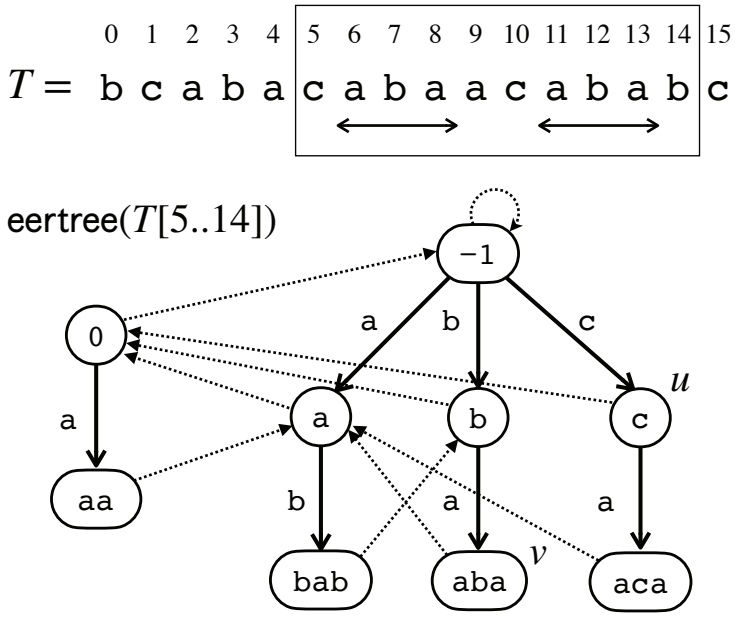

Figure 7.5: Examples for $BegPair_{i,j}(v)$. For string T = bcabacabaacababc and window $[5, 14]$, the eertree($T[5..14]$) is depicted. Consider node v in eertree($T[5..14]$) with $pal(v)$ = aba. The rightmost and the second rightmost occurrences of aba in the window $T[5..14]$ are 11 and 6. Namely, $rm_{5,14}(v) = 11$ and $srm_{5,14}(v) = 6$. Further, $inSL(v) = 0$, and thus, $BegPair_{5,14}(v) = (11, 6)$. Also, for node u in eertree($T[5..14]$) with $pal(u)$ = c, $BegPair_{5,14}(u) = (10, 5)$ since $rm_{5,14}(u) = 10$, $srm_{5,14}(u) = 5$, and $inSL(u) = 0$. When the leftmost character $T[5]$ = c is deleted from the window $T[5..14]$, $srm_{5,14}(u)$ changes to -1. However, $BegPair_{6,14}(u) = BegPair_{5,14}(u) = (10, 5)$ is allowed since $BegPair_{6,14}(u).second = 5 < 6$ is a valid value for our invariant. Namely, we do not have to update $BegPair_{6,14}(u)$ explicitly when deleting the leftmost character $T[5]$ from $T[5..14]$.

$rm_{i,j}(v)$ and $BegPair_{i,j}(v).second = srm_{i,j}(v)$ if $inSL(v) = 0$, and let their values be -1 or some occurrence of $pal(v)$ in $T[0..j]$ otherwise. Namely, $BegPair_{i,j}(v)$ stores the rightmost and second rightmost occurrences of $pal(v)$ in $T[i..j]$ when $inSL(v) = 0$, if such occurrences exist. Otherwise, it temporarily stores some pair of integers, however, it will never be referred to in our algorithms. In other words, we employ a kind of lazy maintenance of the rightmost and second rightmost occurrences of $pal(v)$ in $T[i..j]$ that suffices for our purpose. See Fig. 7.5 for an example of $BegPair_{i,j}(v)$.

The next lemma states that given a node v, we can determine if $pal(v)$ is unique or not by checking the incoming suffix links of v and $BegPair_{i,j}(v)$.

Lemma 7.3. *Let v be any node in* eertree($T[i..j]$)*. Then, $pal(v)$ is unique in $T[i..j]$ if and only*

Algorithm 1 $update\_bp(v, x)$.

Require: Node v, and a starting position x of $pal(v)$.

Ensure: Update $v.bp$ appropriately with respect to the position x.

```
if x > v.bp.first then
    v.bp.second ← v.bp.first
    v.bp.first ← x
else if x > v.bp.second then
    v.bp.second ← x
end if
```

if $inSL(v) = 0$ *and* $BegPair_{i,j}(v).second < i$.

Proof. ($\Rightarrow$) We show the contraposition. There are two cases: (1) If $inSL(v) \neq 0$, then there is a palindromic substring P of $T[i..j]$ with $lps(P) = pal(v)$. Namely, $pal(v)$ is not border-maximal in $T[i..j]$, and thus, $pal(v)$ is not unique in $T[i..j]$. (2) If $inSL(v) = 0$ and $BegPair_{i,j}(v).second = srm_{i,j}(v) \geq i$, then $pal(v)$ occurs at least twice in $T[i..j]$ at positions $BegPair_{i,j}(v).second$ and $BegPair_{i,j}(v).first$.
($\Leftarrow$) For the sake of contradiction, we assume that $pal(v)$ is not unique in $T[i..j]$. Then, by the definition of srm, $i \leq srm_{i,j}(v) \leq j$. However, since $inSL(v) = 0$, $srm_{i,j}(v) = BegPair_{i,j}(v).second < i$, a contradiction. □

Next, we introduce our algorithms to maintain $prefPal$ and $BegPair$ for a sliding window, which utilizes combinatorial properties shown in Section 7.2.

7.3.2 Maintaining the Auxiliary Data Structures

First, in Algorithm 1, we show subroutine $update\_bp$ which updates the member variable $v.bp$ of a given node v where $v.bp$ must be kept equal to $BegPair_{i_t,j_t}(v)$ at each time t. It will be called in the algorithms that we show later.

Next, we show our algorithms for updating data structures when we slide the given window. When the leftmost character $T[i-1]$ is deleted from $T[i-1..j-1]$, our data structures are updated by Algorithm 2. Also, when a character $T[j]$ is appended to $T[i..j-1]$, our data structures are updated by Algorithm 3.

Time Complexities. Clearly, Algorithm 1 runs in constant time. In Algorithm 2, all lines except for Line 13 can be processed in constant time. Thus, the total running time of Algorithm 2

Algorithm 2 Update $BegPair$ and $prefPal$ when the first leftmost character is deleted.

Require: $lpsuf = node(lps(T[i-1..j-1]))$, and $v.bp = BegPair_{i-1,j-1}(v)$ for each node v in $\mathsf{eertree}(T[i-1..j-1])$.

Ensure: $lpsuf = node(lps(T[i..j-1]))$, and $v.bp = BegPair_{i,j-1}(v)$ for each node v in $\mathsf{eertree}(T[i..j-1])$.

1: $lppref \leftarrow prefPal[i-1]$
2: **if** $lppref = lpsuf$ **then**
3: $\quad lpsuf \leftarrow slink(lpsuf)$
4: **end if**
5: $q \leftarrow slink(lppref)$
6: $inSL(q) \leftarrow inSL(q) - 1$
7: $x \leftarrow i - 1 + len(lppref) - len(q)$
8: $update\_bp(q, x)$
9: **if** $len(q) > len(prefPal[x])$ **then**
10: $\quad prefPal[x] = q$
11: **end if**
12: **if** $inSL(lppref) = 0$ and $lppref.bp.second < i - 1$ **then**
13: $\quad$ Remove node $lppref$ from the eertree
14: **end if**

is dominated by Line 13, i.e., $O(\log \sigma')$. In Algorithm 3, the first four lines can be processed in amortized $O(\log \sigma')$ time by using the online construction algorithm [50]. Also, the remaining lines can be processed in constant time, and thus, the total running time of Algorithm 3 is amortized $O(\log \sigma')$.

Correctness. First, it is clear that Algorithm 1 runs correctly. Next, let us consider the correctness of Algorithm 2. Let us first consider a special case when the window $T[i-1..j-1]$ itself is a palindrome. Then, we need to update $lpsuf$, which will be used in Algorithm 3. Lines 2–3 of Algorithm 2 captures such a case. Next, we show that $BegPair$ for all nodes are updated correctly. By the invariant of $BegPair$, it suffices to update $v.bp$ for every node v where $inSL(v) = 0$, i.e., $pal(v)$ is border-maximal. Let q be the node corresponding to the longest palindromic suffix of $lppref = lpp(T[i-1..j-1])$. Then, it suffices to update $q.bp$ since the node q is the only candidate for a node whose corresponding palindrome to be border-maximal *just* in this step. Thus, we update only $q.bp$ in Lines 5–8, if it is needed. Further, we show

Algorithm 3 Update $BegPair$ and $prefPal$ when a character is appended.

Require: $lpsuf = node(lps(T[i..j-1]))$, $T[j]$, and $v.bp = BegPair_{i,j-1}(v)$ for each node v in $\mathsf{eertree}(T[i..j-1])$.

Ensure: $lpsuf = node(lps(T[i..j]))$, and $v.bp = BegPair_{i,j}(v)$ for each node v in $\mathsf{eertree}(T[i..j])$.

1: $lpsuf \leftarrow node(lps(T[i..j]))$
2: **if** $lpsuf$ is not in $\mathsf{eertree}(T[i..j-1])$ **then**
3: Add new node $lpsuf$ to the eertree
4: **end if**
5: $y \leftarrow j - len(lpsuf) + 1$
6: $update\_bp(lpsuf, y)$
7: $prefPal[y] \leftarrow lpsuf$

that $prefPal$ is also updated correctly. By Lemma 7.2, the longest palindromic prefix of a window must be the longest palindromic suffix of either some window or the longest palindromic prefix of some window. The palindrome $pal(q)$ is the only one that is to be such a palindrome *just* in this step. Thus, $prefPal[x]$ is the only candidate which may be updated in this step where x is the starting position of the occurrence of $pal(q)$ that is the longest palindromic suffix of $lpp(T[i-1, j-1])$. Therefore, it suffices to update $prefPal[x]$ and update it if necessary (Lines 9–11). Line 12 determines the uniqueness of $lpp(T[i-1..j-1])$ correctly by using Lemma 7.3, and if it is unique, then the corresponding node $lppref$ is removed (in Line 13).

Finally, consider the correctness of Algorithm 3. When a character is appended, we first check the new longest palindromic suffix, and create a new node corresponding to the palindrome if necessary. These procedures in Lines 1–4 are correctly performed by running the online construction algorithm [50]. Let y be the starting position of the longest palindromic suffix of the window $T[i..j]$. The palindrome $lps(T[i..j])$ is the only candidate for a palindrome to be border-maximal *just* in this step, and thus, it suffices to update $lpsuf.bp$ in this step. Also, $lpsuf$ is the only candidate for the node that we need to newly store into $prefPal$ in this step. At this moment, $lpsuf$ is clearly the longest palindrome starting at position y. Thus, we set $prefPal[y] \leftarrow lpsuf$ (Line 7).

To summarize this section, we obtain the following theorem.

Theorem 7.1. *We can maintain eertrees for a sliding window in a total of $O(n \log \sigma')$ time using $O(d') + d$ space where $d' \leq d$ is the maximum number of distinct palindromes in all windows.*

Proof. When a character is appended to the right end of the window, we update the eertree

itself by applying the online algorithm [50], and update our auxiliary data structures by using Algorithm 3. When the leftmost character is deleted from the window, we update the eertree and the auxiliary data structures by using Algorithm 2. The total running time is $O(n \log \sigma')$. The space usage is $O(d')$ words for the original eertree and the auxiliary member variable $v.bp$ for each node v of the eertree, plus d words for the array $prefPal[0..d-1]$. □

By applying a subtle modification to the above algorithm, we obtain another variant of the algorithm (Theorem 7.2 below) which is faster than Theorem 7.1 when $d'\sigma < n \log \sigma'$, but using additional $(d'+1)\sigma$ space.

Theorem 7.2. *We can maintain eertrees for a sliding window in a total of $O(n + d'\sigma)$ time using $(d'+1)\sigma + O(d') + d \in O(d\sigma)$ space.*

Proof. In the original eertrees, each node stores a binary search tree to maintain branches dynamically. Instead, we use an array of integers of size σ, which allows us to add, delete, and search for a node pointer (i.e., edge) labeled by a given character in constant time. Thus, the $\log \sigma'$ factor in our time complexity can be removed. On the other hand, we need $\sigma + O(1)$ space to represent each node object, and $\Theta(\sigma)$ time to initialize it. If we naively initialize such a node object when adding a new node, the total time complexity increases to $O(n\sigma)$. However, we can *reuse* node objects that had been removed when deleting a character since such removed nodes, and new nodes to be added are leaves, i.e., they do not have any child (Lemma 7.1). Thus, by reusing node objects, we do not need to initialize an array of size σ when adding a new leaf node. The total number of node objects to initialize is $d'+1$, and it costs $O(d'\sigma)$ total time to initialize them. □

7.4 Applications of Eertrees for a Sliding Window

In this section, we apply our sliding-window eertree algorithm of Section 7.3 to computing minimal unique palindromic substrings and minimal absent palindromic words for a sliding window.

7.4.1 Computing Minimal Unique Palindromic Substrings for a Sliding Window

Now, we show Lemma 7.4 which states a relationship between eertrees and MUPSs. Then, in Lemma 7.5, we show that all MUPS can be computed using eertrees in an offline manner.

Lemma 7.4. *A string w is a MUPS of T if and only if there is a node v in* $\mathsf{eertree}(T)$ *such that $pal(v) = w$, $pal(v)$ is unique in T and $pal(u)$ is repeating in T, where u is the parent of v.*

Proof. ($\Rightarrow$) Since w is a MUPS of T, it is clear that there is a node v such that $pal(v) = w$ and it is unique in T. Also, since $pal(v) = w \neq \varepsilon$, v has the parent u, which represents the string $w[1..|w| - 2]$. By the definition of MUPS, $pal(u) = w[1..|w| - 2]$ is repeating in T. ($\Leftarrow$) Since the palindrome $pal(v) = w$ is unique in T and $pal(u) = w[1..|w| - 2]$ is repeating in T, w is a MUPS of T. □

Lemma 7.5. *Given* $\mathsf{eertree}(T)$*, we can compute* MUPS_T *in* $O(|\mathsf{DPal}(T)|)$ *time.*

Proof. Given a node v, we can detect whether $pal(v)$ is a MUPS or not in constant time by Lemma 7.4. Also, the starting position of a palindrome $pal(v)$, which is unique in T is stored in $v.bp.first$. Therefore, we can compute MUPS_T by a single traversal on $\mathsf{eertree}(T)$. □

Moreover, we can efficiently maintain MUPSs for a sliding window.

Lemma 7.6. *We can maintain the set of MUPSs for a sliding window in a total of $O(n \log \sigma')$ time using $O(d)$ space.*

Proof. In addition to the eertree data structure described in Section 7.3, we add 1-bit information ismups into each node. This bit ismups is set to 1 if the node corresponds to a MUPS and to 0 otherwise. We first consider to delete the leftmost character $T[i-1]$ from $T[i-1..j-1]$. In this case, only prefixes of $T[i-1..j-1]$ are those whose number of occurrences in the sliding window change. We check the nodes corresponding to the longest and the second longest palindromic prefixes, and update ismups of them accordingly. We do not need to care about other palindromic prefixes since they must be repeating in $T[i..j-1]$. Symmetrically, we can easily maintain ismups in the case when appending a character $T[j]$ to $T[i..j-1]$. □

7.4.2 Computing Minimal Absent Palindromic Words for a Sliding Window

For a relation between MAPWs and eertrees, the following lemmas hold.

Lemma 7.7. *For any non-empty string $w \in \Sigma^*$, w is a MAPW of a string T if and only if there is a node u in* $\mathsf{eertree}(T)$ *such that $pal(u) = w[1..|w| - 2]$, $len(u) = |w| - 2$, and u does not have an edge labeled by $w[0]$.*

Proof. ($\Rightarrow$) Since w is a MAPW, $w[1..|w|-2]$ is a palindromic substring of T, and thus, there is a node u with $pal(u) = w[1..|w|-2]$ and $len(u) = |w|-2$. Also, since the palindrome w does not occur in T, u does not have an edge labeled by $w[0]$. ($\Leftarrow$) Since u is a node in $\mathsf{eertree}(T)$, the string $pal(u) = w[1..|w|-2]$ occurs in T. Also, since u does not have an edge labeled by $w[0]$, the string w does not occur in T. Thus, w is a MAPW of T. □

Lemma 7.8. *Let* MAPW_T *be the set of MAPWs of* T*. Then* $|\mathsf{MAPW}_T| = 2\sigma + (|\mathsf{DPal}(T)| - 1)(\sigma - 1)$*. Also, given* $\mathsf{eertree}(T)$*,* MAPW_T *can be computed in* $O(|\mathsf{DPal}(T)|\sigma)$ *time.*

Proof. We prove $|\mathsf{MAPW}_T| = 2\sigma + (|\mathsf{DPal}(T)| - 1)(\sigma - 1)$ by induction. If $S = \varepsilon$, then $\mathsf{DPal}(T) = \{\varepsilon\}$ and MAPW_T consists of a and aa for each character $a \in \Sigma$. Thus, $|\mathsf{MAPW}_\varepsilon| = 2\sigma = 2\sigma + (|\mathsf{DPal}(\varepsilon)| - 1)(\sigma - 1)$ holds. We assume that $|\mathsf{MAPW}_T| = 2\sigma + (|\mathsf{DPal}(T)| - 1)(\sigma - 1)$ holds for any string T of length $k \geq 0$. Then consider any string T' of length $k+1$, and let $T' = Tc$ where $c \in \Sigma$. If $\mathsf{DPal}(T) = \mathsf{DPal}(Tc)$, then it is clear that $\mathsf{MAPW}_{Tc} = \mathsf{MAPW}_T$, and hence, $|\mathsf{MAPW}_{Tc}| = |\mathsf{MAPW}_T| = 2\sigma + (|\mathsf{DPal}(T)| - 1)(\sigma - 1) = 2\sigma + (|\mathsf{DPal}(Tc)| - 1)(\sigma - 1)$. Otherwise, it is known that $\mathsf{DPal}(T) \setminus \mathsf{DPal}(Tc) = \emptyset$ and $\mathsf{DPal}(Tc) \setminus \mathsf{DPal}(T) = \{lps(Tc)\}$ [16]. Let $P = lps(Tc)$ be the only palindrome in $\mathsf{DPal}(Tc) \setminus \mathsf{DPal}(T)$. Since P is absent from T and $P[1..|P|-2]$ occurs in T (at least as a suffix of T), P is in MAPW_T. Also, P occurs in Tc and aPa is absent from Tc for each character a, and hence, $|\mathsf{MAPW}_{Tc}| = |\mathsf{MAPW}_T| + (\sigma - 1)$ holds. Therefore, $|\mathsf{MAPW}_{Tc}| = |\mathsf{MAPW}_T| + (\sigma - 1) = 2\sigma + |\mathsf{DPal}(T)|(\sigma - 1) = 2\sigma + (|\mathsf{DPal}(Tc)| - 1)(\sigma - 1)$.

Next, we prove the time complexity for computing MAPWs with $\mathsf{eertree}(T)$. By Lemma 7.7, an ordinary node v does not have an edge labeled by c if and only if $c\,pal(v)c \in \mathsf{MAPW}_T$. Also, a MAPW of length 1 is an absent character. Thus, it is easy to see that the information of all MAPWs can be computed by traversing $\mathsf{eertree}(T)$ only once. □

Also, we can maintain MAPWs for a sliding window by applying Theorem 7.2.

Corollary 7.1. *We can maintain the set of MAPWs for a sliding window in a total of* $O(n + d\sigma)$ *time using* $O(d\sigma)$ *space.*

7.5 Conclusions and Future Work

In this chapter, we studied the problem of computing MUPSs for a sliding window of size d over a given string T of length n. For the sake of computing MUPSs efficiently, we first considered the problem of maintaining the palindromic trees for a sliding window. We then proposed an

algorithm which maintains eertrees for a sliding window in a total of $O(n \log \sigma')$ time using $O(d') + d$ space, where $d' \leq d$ be the maximum number of distinct palindromes in all windows. Also, we give an alternative eertree construction algorithm for a sliding window that runs in $O(n + d\sigma)$ time with $(d+2)\sigma + O(d)$ space. Furthermore, we proposed (i) an algorithm which maintains MUPSs for a sliding window in a total of $O(n \log \sigma')$ time using $O(d') + d$ space, and (ii) an algorithm which maintains MAPWs for a sliding window in a total of $O(n + d\sigma)$ time using $O(d\sigma)$ space.

While the size of the original eertree data structure is $O(d')$, our first algorithm for maintaining eertrees (that runs $O(n \log \sigma')$ time) requires additional d-words of space. This is because we use array $prefPal[0..d-1]$ for maintaining the longest palindromic prefixes of the windows. Thus, we have an open question: Can we design a sliding-window algorithm for computing eertrees *without* using any $\Omega(d)$-space data structure?

Chapter 8

Conclusions

In this thesis, we studied the problems of computing unique substrings and proposed space-efficient data structures based on combinatorial properties on unique substrings.

In Chapter 3, we showed the tight upper bound $(3n - 1)/2$ for the maximum number of substrings which can be a SUS for some text position in a string of length n. We also introduced the notion of non-trivial SUS. Then, we showed that the number of non-trivial SUSs in a string is less than $2n$, and this upper bound is asymptotically tight. These are the first non-trivial results for combinatorial properties on the shortest unique substring problems.

In Chapter 4, we addressed the SUS problem in the case where the string is given in run-length encoding (RLE). We first showed that the number m of MUSs of a string is less than twice the size r of the RLE string, and the upper bound is tight. Based on the combinatorial results, we proposed a data structure of size $O(r)$ such that we can answer any interval SUS query in $O(\sqrt{\log r / \log\log r} + k)$ time where k is the number of SUSs to output. Since $r \leq n$ always holds, the size of our data structure is not worse than $O(n)$. Notably, our method can be sub-linear to n when the input string is well-compressible by RLE.

In Chapter 5, we also proposed an alternative data structure of size $4n + o(n)$ bits for the interval SUS problem by utilizing succinct data structures. Furthermore, we gave a smaller data structure of size $2.6n + o(n)$ bits for the point SUS problem. These $O(n)$ bits data structures can be constructed in $O(n)$ time, and the working space is also small.

In Chapter 6, we considered the problem of maintaining the set of MUSs in a semi-dynamic string, that is, we can append a character to the right-end of the string and delete the leftmost character of the string. We proposed an algorithm running in amortized $O(\log \sigma)$ time per operation, using $O(n)$ space. Our method immediately yields a sliding-window algorithm for maintaining MUSs in the windows that runs in a total of $O(n \log \sigma)$ time using $O(d)$ space

where d is the size of the window.

In Chapter 7, we considered the problem of maintaining the set of MUPSs in a semi-dynamic string. For the sake of maintaining MUPSs, we first proposed an algorithm for maintaining the palindromic tree (a.k.a. eertree) of a semi-dynamic string that runs in amortized $O(\log \sigma)$ time per operation, using $O(n)$ space. Then, by utilizing the result, we obtained an algorithm for maintaining MUPSs in a semi-dynamic string with the same running time and space.

Bibliography

[1] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, Apr. 1996.

[2] A. Apostolico, G. M. Landau, and S. Skiena. Matching for run-length encoded strings. *J. Complex.*, 15(1):4–16, 1999.

[3] O. Arbell, G. M. Landau, and J. S. Mitchell. Edit distance of run-length encoded strings. *Information Processing Letters*, 83(6):307 – 314, 2002.

[4] G. Badkobeh, G. Fici, S. Kroon, and Z. Lipták. Binary jumbled string matching for highly run-length compressible texts. *Information Processing Letters*, 113(17):604 – 608, 2013.

[5] H. Bannai, T. Gagie, S. Inenaga, J. Kärkkäinen, D. Kempa, M. Piatkowski, and S. Sugimoto. Diverse palindromic factorization is NP-complete. *Int. J. Found. Comput. Sci.*, 29(2):143–164, 2018.

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

[7] H. Bunke and J. Csirik. An algorithm for matching run-length coded strings. *Computing*, 50(4):297–314, 1993.

[8] H. Bunke and J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, Apr. 1995.

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

[10] K.-Y. Chen, P.-H. Hsu, and K.-M. Chao. Efficient retrieval of approximate palindromes in a run-length encoded string. *Theoretical Computer Science*, 432:28 – 37, 2012.

[11] D. R. Clark. *Compact Pat Trees*. PhD thesis, 1998. UMI Order No. GAXNQ-21335.

[12] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Communications*, 32(4):396–402, 1984.

[13] M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis, and Y. Ramusat. Absent words in a sliding window with applications. *Inf. Comput.*, 270, 2020.

[14] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Data compression using antidictionaries. *Proceedings of the IEEE*, 88(11):1756–1768, 2000.

[15] P. Davoodi, R. Raman, and S. R. Satti. Succinct representations of binary trees for range minimum queries. In *Proceedings of the 18th Annual International Computing and Combinatorics Conference (COCOON)*, pages 396–407, 2012.

[16] X. Droubay, J. Justin, and G. Pirillo. Episturmian words and some constructions of de Luca and Rauzy. *Theor. Comput. Sci.*, 255(1-2):539–553, 2001.

[17] E. R. Fiala and D. H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989.

[18] G. Fici, T. Gagie, J. Kärkkäinen, and D. Kempa. A subquadratic algorithm for minimum palindromic factorization. *J. Discrete Algorithms*, 28:41–48, 2014.

[19] J. Fischer, T. I, D. Köppl, and K. Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.

[20] Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda. Computing DAWGs and minimal absent words in linear time for integer alphabets. In *Proceedings of the 41st International Symposium on Mathematical Foundations of Computer Science (MFCS '16)*, pages 38:1–38:14, 2016.

[21] A. Ganguly, W.-K. Hon, R. Shah, and S. V. Thankachan. Space-time trade-offs for finding shortest unique substrings and maximal unique matches. *Theoretical Computer Science*, 700:75–88, 2017.

[22] K. Goto and H. Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proceedings of 2014 Data Compression Conference (DCC)*, pages 163–172. IEEE, 2014.

[23] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[24] R. Groult, É. Prieur, and G. Richomme. Counting distinct palindromes in a word in linear time. *Inf. Process. Lett.*, 110(20):908–912, 2010.

[25] B. Haubold, N. Pierstorff, F. Möller, and T. Wiehe. Genome comparison without alignment using shortest unique substrings. *BMC bioinformatics*, 6(1):123, 2005.

[26] W. Hon, S. V. Thankachan, and B. Xu. In-place algorithms for exact and approximate shortest unique substring problems. *Theor. Comput. Sci.*, 690:12–25, 2017.

[27] X. Hu, J. Pei, and Y. Tao. Shortest unique queries on strings. In *Proceedings of String Processing and Information Retrieval (SPIRE)*, pages 161–172, 2014.

[28] A. M. İleri, M. O. Külekci, and B. Xu. A simple yet time-optimal and linear-space algorithm for shortest unique substring queries. *Theoretical Computer Science*, 562:621–633, 2015.

[29] L. Ilie and W. F. Smyth. Minimum unique substrings and maximum repeats. *Fundam. Inform.*, 110(1-4):183–195, 2011.

[30] H. Inoue, Y. Nakashima, T. Mieno, S. Inenaga, H. Bannai, and M. Takeda. Algorithms and combinatorial properties on shortest unique palindromic substrings. *J. Discrete Algorithms*, 52-53:122–132, 2018.

[31] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[32] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proceedings of Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009.

[33] D. Kempa. Optimal construction of compressed indexes for highly repetitive texts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1344–1357, 2019.

[34] J. W. Kim, A. Amir, G. M. Landau, and K. Park. Computing similarity of run-length encoded strings with affine gap penalty. *Theoretical Computer Science*, 395(2–3):268 – 282, 2008. SAIL – String Algorithms, Information and Learning: Dedicated to Professor Alberto Apostolico on the occasion of his 60th birthday.

[35] D. Köppl. Computing lexicographic parsings. Technical report, TU Dortmund, 2019.

[36] D. Kosolobov, M. Rubinchik, and A. M. Shur. Finding distinct subpalindromes online. In *Proceedings of the Prague Stringology Conference 2013*, pages 63–69, 2013.

[37] N. J. Larsson. Extended application of suffix trees to data compression. In *Proceedings of the 6th Data Compression Conference (DCC '96)*, pages 190–199, 1996.

[38] Z. Li, J. Li, and H. Huo. Optimal in-place suffix sorting. In *Proceedings of 2018 Data Compression Conference (DCC)*, pages 422–422, 2018.

[39] J.-J. Liu, G.-S. Huang, and Y.-L. Wang. A fast algorithm for finding the positions of all squares in a run-length encoded string. *Theoretical Computer Science*, 410(38–40):3942 – 3948, 2009.

[40] Mäkinen, Ukkonen, and Navarro. Approximate matching of run-length compressed strings. *Algorithmica*, 35(4):347–369, 2003.

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

[42] 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.

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

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

[45] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 408–424, 2017.

[46] J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

[47] T. Ohno, Y. Takabatake, T. I, and H. Sakamoto. A faster implementation of online run-length Burrows-Wheeler transform. In *Proceedings of Combinatorial Algorithms - 28th International Workshop, IWOCA 2017*, pages 409–419, 2017.

[48] T. Ota and H. Morita. On a universal antidictionary coding for stationary ergodic sources with finite alphabet. In *Proceedings of the International Symposium on Information Theory and its Applications (ISITA '14)*, pages 294–298, 2014.

[49] J. Pei, W. C. Wu, and M. Yeh. On shortest unique substring queries. In *Proceedings of IEEE 29th International Conference on Data Engineering (ICDE)*, pages 937–948, 2013.

[50] M. Rubinchik and A. M. Shur. EERTREE: an efficient data structure for processing palindromes in strings. *Eur. J. Comb.*, 68:249–265, 2018.

[51] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 225–232, 2002.

[52] M. Senft. Suffix tree for a sliding window: An overview. In *WDS '05*, pages 41–46, 2005.

[53] M. Takeda. *Encyclopedia of algorithms*, chapter "Compressed Pattern Matching", pages 171–174. Springer US, 2008.

[54] Y. Tamakoshi, K. Goto, S. Inenaga, H. Bannai, and M. Takeda. An opportunistic text indexing structure based on run length encoding. In *Proc. CIAC 2015*, pages 390–402, 2015.

[55] K. Tsuruta, S. Inenaga, H. Bannai, and M. Takeda. Shortest unique substrings queries in optimal time. In *Proceedings of SOFSEM 2014: Theory and Practice of Computer Science*, pages 503–513, 2014.

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

[57] K. Watanabe, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Fast algorithms for the shortest unique palindromic substring problem on run-length encoded strings. *Theory Comput. Syst.*, 2020.

[58] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.