Factorizing Strings into Combinatorial Objects

杉本, 志穂

https://doi.org/10.15017/1928634

Factorizing Strings into Combinatorial Objects

Shiho Sugimoto

August, 2017

Abstract

String factorization is a task of factorizing a string into a sequence of non-empty strings under given constraints. The size of factorization is defined to be the number of factors in it. For example, the Lyndon factorization has the constraints that each factor is a Lyndon word and the factors are arranged in the lexicographically non-increasing order. On the other hand, the Lempel-Ziv 77 (LZ77) factorization, the core of LZ77 compression, is a smallest-sized factorization such that each factor has a previous occurrence.

It is known that the sizes of the Lyndon factorization and the LZ77 factorization of a string w are lower bounds on the size of grammar that generates w in the grammar-based compression. Moreover, these factorizations can be used as a means of accelerating several string processing algorithms. Studies on string factorization can thus contribute not only to Combinatorics on Strings but also to String Algorithms. Many kinds of string factorizations have been introduced and efficient factorization algorithms have been developed for them.

Kolpakov and Kucherov proposed a variant of the LZ77 factorization, called the reversed LZ factorization (Theoretical Computer Science, 410(51), 2009). We present an online algorithm that computes the reversed LZ factorization of a given string w of length n in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space, where $\sigma \leq n$ is the alphabet size. Also, we introduce a new variant, called the reversed LZ factorization with self-references, and present two online algorithms to compute this variant, in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space, and in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space.

A *palindromic factorization* of a string w is a factorization of w into palindromic substrings of w. We present an online $O(n \log n)$ time $O(n)$ space algorithm to compute a smallest-sized palindromic factorization of every prefix of w, where n is the length of a given string w. We then show how to extend this algorithm to compute a smallest-sized maximal palindromic factorizations of every prefix of w, where the factors are maximal palindromes (non-extensible palindromic substring) of the prefix, in $O(n \log n)$ time and $O(n)$ space, in an online manner.

We also present an online algorithm that computes a smallest-sized palindromic cover of w in $O(n)$ time using $O(n)$ space.

We also define a restricted variant of palindromic factorization, called the diverse palindromic factorization, where the factors are distinct from each other. We prove that the existence problem of the diverse palindromic factorization is NP-complete.

A *closed string* is a string with a proper substring that occurs in the string as a prefix *and* a suffix, but not elsewhere. Closed strings were introduced by Fici (WORDS 2011) as objects of combinatorial interest in the study of Trapezoidal and Sturmian words. We consider algorithms for computing closed factors (substrings) in strings, and in particular for greedily factorizing a string into a sequence of longest closed factors. We describe an algorithm for this problem that uses linear time and space. We then consider the related problem of computing, for every position in the string, the longest closed factor starting at that position. We present a simple algorithm for the problem that runs in $O(n \log n / \log \log n)$ time.

Two strings x and y are said to be Abelian equivalent if x is a permutation of y, or vice versa. If a string z satisfies $z = xy$ with x and y being Abelian equivalent, then z is said to be an *Abelian square*. If a string w can be factorized into a sequence $v_1, \ldots, v_s$ of strings such that v_1, $\ldots$, v_{s-1} are all Abelian equivalent and v_s is a substring of a permutation of v_1, then w is said to have a *regular Abelian period* (p, t) where $p = |v_1|$ and $t = |v_s|$. If a substring $w_1[i..i + l - 1]$ of a string w_1 and a substring $w_2[j..j + l - 1]$ of another string w_2 are Abelian equivalent, then the substrings are said to be a common Abelian factor of w_1 and w_2 and if the length l is the maximum of such then the substrings are said to be a *longest common Abelian factor* of w_1 and w_2. We propose efficient algorithms which compute these Abelian regularities using the *run length encoding (RLE)* of strings. For a given string w of length n whose RLE is of size m, we propose algorithms which compute all Abelian squares occurring in w in $O(mn)$ time, and all regular Abelian periods of w in $O(mn)$ time. For two given strings w_1 and w_2 of total length n and of total RLE size m, we propose an algorithm which computes all longest common Abelian factors in $O(m^2 n)$ time.

Acknowledgements

I really appreciate all the support I received from everyone. I begin by my most grateful thanks for Professor Masayuki Takeda who is my supervisor and the committee chair of this thesis. He gave me the choice to try the PhD course. I learned what research is, what researcher is, and how to enjoy the life as a researcher from what he did. Thanks to him, my life in laboratory could not have been better. I also express my appreciation to Professor Eiji Takimoto, Professor Daisuke Ikeda and Professor Yukiko Yamauchi, who are the members of the committee of this thesis. They gave me some advice to make this thesis better.

I greatly appreciate to Professor Hideo Bannai and Professor Shunsuke Inenaga. They were always my goals far from me. It was a happiness that I always had my role models. They had many big ideas, knowledges and techniques and they had never hesitated to give it us. I am grateful to Dr. Tomohiro I in Kyushu Institute of Technology. He helped me my research when and after he was in Kyushu University. I also appreciate to Professor Eiji Takimoto and Professor Kohei Hatano for their advice in weekly seminars. The discussions with them were always productive.

This research was partly supported JSPS (Japan Society for the Promotion of Science). The results in the thesis were partially published in the Proc. of PSC2013, the Proc. of PSC2014, the Proc. of CPM2014, the Proc. of DLT2015 and the Proc. of IWOCA2017. Also, the journal version of PSC2014 paper was published in 2016 in Discrete Applied Mathematics by ELSEVIER. I am thankful for all editors, committees, anonymous referees, and publishers.

I express my gratitude to co-authors Dr. Golnaz Badkobeh, Dr. Travis Gagie, Professor Costas S. Iliopoulos, Dr. Juha Kärkkäinen, Dr. Dominik Kempa, Dr. Marcin Piatkowski and Dr. Simon J. Puglisi. Especially, I really enjoyed discussing stringology and talking about others with Dr. Golnaz Badkobeh and Dr. Simon J. Puglisi. Additionally, discussing with Professor Maxime Crochemore and Dr. Tatiana Starikovskaya were great time to me. I hope I will write a paper with them.

I am also grateful to the technical staffs of our laboratory, Ms. Sanae Wakita, Ms. Miho Higo and Ms. Akiko Ikeuchi. They always supported and took care of me, not limited to my paperwork . I also express thanks to all of staffs in Department of Informatics, Kyushu University.

Last, but not least, I express my lots of thanks and lots of loves to my family.

Contents

Chapter 1

Introduction

1.1 Background

Combinatorics is an area of Discrete Mathematics, which studies how to count or enumerate certain combinatorial objects. Combinatorics on Words, branching from Combinatorics, focuses on words or strings, i.e., sequences of symbols. Examples of combinatorial string objects are squares, repetitions (i.e., periodic strings), palindromes, Lyndon words, closed strings, and so on.

A string factorization is a research topic in Combinatorics on Words, and has received special concerns. It is a task of factorizing a given string into combinatorial objects. An example is the Lyndon factorization introduced by Chen, Fox and Lyndon [17] in 1958. The goal of studies on string factorization in this area is mainly to discover algebraic properties on strings. Little attention has been paid to development of algorithms for factorizing strings or analysis of their complexities in this area.

On the other hand, Stringology is an area of Theoretical Computer Science, which studies algorithms and data structures for processing string data efficiently as well as algebraic properties on combinatorial string objects. Algorithmic aspects of string factorization have been a subject of active research in Stringlogy. For the Lyndon factorization, Duval [27] revealed in 1983 the property that a sequence of longest Lyndon words corresponds to the Lyndon factorization, and presented a simple linear-time algorithm based on this property. Efficient parallel algorithms for the Lyndon factorization are also known [7, 25]. Recently, Lyndon factorization algorithms over compressed string were proposed [41, 42].

The Lyndon factorization is of a mathematical interest but it has several applications, especially related to data compression. It is used in a bijective variant of Burrows-Wheeler trans-

form [57, 37][1]. In the context of grammar-based compression, it is shown in [42] that the size of Lyndon factorization of a string gives a lower bound on the size of a smallest grammar that generates it uniquely. Examples of applications not related to data compression are a digital geometry algorithm [14] and a public key cryptosystem [69]. Very recently, it turns out that the Lyndon factorization can be used as a preprocessing step in computation of runs (maximal repetitions) within a given string [54, 36, 21].

The goal of this thesis is to develop efficient algorithms for existing or newly introduced string factorization problems, or to show their intractability.

1.2 Previous work on string factorization

First, we mention a class of string factorizations which were originally intended for data compression. Ziv and Lempel [78] proposed the LZ77 factorization as a tool of data compression. It plays a central role in efficient string processing algorithms [52, 28], in compressed full text indices [56], and in an approximation algorithm to the smallest grammar problem [67].

Due to its importance, efficient algorithms for computing the LZ77 factorization have been proposed, both in offline manner and in online manner. In the offline setting where the string is static, there exist efficient algorithms to compute the LZ77 factorization of a given string w of length n, running in $O(n)$ time and using $O(n \log n)$ bits of space, assuming an integer alphabet [19]. See [1] for a survey, and [48, 38, 47, 46] for more recent results in this line of research.

In the online setting where new characters may be appended to the end of the string, Okanohara and Sadakane [66] gave an algorithm that runs in $O(n \log^3 n)$ time using $n \log \sigma + o(n \log \sigma) + O(n)$ bits of space, where σ is the size of the alphabet. Later, Starikovskaya [70] proposed an algorithm running in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space, assuming $\frac{\log_\sigma N}{4}$ characters are packed in a machine word. Yamamoto et al. [76] developed an online LZ77 factorization algorithm running in $O(n \log n)$ time using $O(n \log \sigma)$ bits of space.

Some variants of the LZ77 factorization also exist; the LZ-End factorization [55] allows faster random access in compressed strings, and the *reversed* LZ factorization [53, 71] is useful for finding gapped palindromes in strings.

Ziv and Lempel [79] also proposed the LZ78 factorization, and later Welch [75] proposed its variant called the LZW factorization. These factorizations are intended for data compression.

[1] The Burrows-Wheeler transform is a core of the compression tool `bzip2`.

Also, they can be used as a means of accelerating computation of alignments of two strings [22]. The LZ78/LZW factorization of a string of length n over an alphabet of size σ can be computed in $O(n \log \sigma)$ time. A more efficient algorithm for computing the LZ78/LZW factorization was proposed by Jansson et al. [45].

Next, we mention several string factorizations not related to data compression. The smallest-sized maximal palindromic factorization problem introduced by Alatabbi et al. [3] is to find a factorization of a given string into factors that are maximal palindromes (non-extensible palindromic substrings) of smallest size. They presented in [3] an offline $O(n)$-time algorithm.

Dumitran et al. [26] introduced two factorization problems: the square factorization problem and the repetition factorization problem, which are defined as the problems of finding *any* factorization of a given string into squares and finding *any* factorization of a given string into runs (maximal repetitions), respectively. For the square factorization problem, they presented in [26] an $O(n \log n)$ time solution. Matsuoka et al. [63] proposed an improved algorithm that runs in $O(n)$ time. They also proposed in [63] algorithms that compute square factorizations of smallest/largest size in $O(n \log n)$ time using $O(n)$ space, assuming an integer alphabet.

For the repetition factorization problem, Dumitran et al. [26] presented an $O(n)$ time algorithm. Inoue et al. [44] proposed algorithms that compute repetition factorizations of smallest/largest size in $O(n \log n)$ time using $O(n)$ space for a general ordered alphabet.

Table 1.1 summarizes known string factorizations and the complexity of existing algorithms for them.

1.3 Our contributions

In this section, we state the problems we consider in this thesis and explain related works of the problems and outline of our solution.

1.3.1 Reversed LZ factorization

One of the most well-studied factorizations is the LZ77 factorization we mentioned above. In this thesis we consider the *reversed* LZ factorization problem introduced by Kolpakov and Kucherov [53], which is used as a basis of computing gapped palindromes. In the online setting, the reversed LZ factorization can be computed in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space, utilizing the algorithm by Blumer et al. [12].

Table 1.1: Several string factorization problems are listed with complexity of existing solutions, where n is the input string length and σ is the alphabet size.

problem	constraint	complexity
Lyndon factozation [17]	each factor is a Lyndon word and the factors are arranged in lexicographically non-decreasing order.	$O(n)$ time $O(n)$ space [27]
LZ77 factorization [78]	each factor is the right maximal substring that has a previous occurrence.	$O(n)$ time $O(n \log n)$ bits of space [19]
reversed LZ factorization [53]	each factor is the right maximal substring of which reversal has a previous occurence.	$O(n \log \sigma)$ time $O(n \log n)$ bits of space [53]
smallest sized maximal palindromic factorization [3]	each factor is a maximal palindrome.	$O(n)$ time $O(n)$ space [3]
square factorization [26]	each factor is a square.	$O(n)$ time $O(n)$ space [63]
smallest/largest sized square factorization [63]	each factor is a square.	$O(n \log n)$ time $O(n)$ space [63]
repetition factorization [26]	each factor is a repetition.	$O(n)$ time $O(n)$ space [26]
smallest/largest sized repetition factorization [44]	each factor is a repetition.	$O(n \log n)$ time $O(n)$ space [44]

We present a space-efficient solution to compute reversed LZ factorization, which requires only $O(n \log \sigma)$ bits of working space with slightly slower $O(n \log^2 n)$ running time.

We also introduce a new, self-referencing variant of the reversed LZ factorization, and propose two online algorithms; the first one runs in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space, and the second one in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space. A key to achieve such complexity is efficient online computation of the longest suffix palindrome for each prefix of the string w.

As an independent interest, we consider the relationship between the number of factors in the reversed LZ factorization of a string w, and the size of the smallest grammar that generates only w. It is known that the number of factors in the LZ77 factorization of w is a lower bound of the smallest grammar for w [67]. We show that, unfortunately, this is not the case with the reversed LZ factorization with or without self-references.

1.3.2 Palindromic factorization

In 2013, Alatabbi et al. [3] introduced a new kind of factorization problem, called the *smallest-sized maximal palindromic factorization problem*. A factorization of string w is said to be a

maximal palindromic factorization of w if every factor in the factorization is a maximal palindrome in w. They presented an offline $O(n)$-time algorithm to compute a smallest-sized maximal palindrome factorization of a given string of length n.

In this thesis, we introduce yet another kind of factorization problem, called the *smallest-sized palindromic factorization problem*. A factorization of a string w is said to be a palindromic factorization of w if every factor is a palindrome (not necessarily maximal). We present an online $O(n \log n)$-time $O(n)$-space algorithm to compute a smallest-sized palindromic factorization of a given string w of length n. In addition, we show how to extend this algorithm to obtain an online $O(n \log n)$-time $O(n)$-space algorithm to compute a smallest-sized maximal palindromic factorization of w. We achieve the $O(n \log n)$-time bound in our solutions using combinatorial properties of palindromic suffixes of strings. We remark that the algorithm of Alattabi et al. [3] is offline, and a naïve extension of their algorithm to the online scenario leads to an $O(n^2)$-time bound.

Also, we consider the problem of covering a given string with fewest palindromes. We show how to compute such covers in $O(n)$ time in an offline fashion, and later describe how to extend it to an online $O(n)$-time algorithm. Both of the algorithms use $O(n)$ space. This solves an open problem of Alatabbi et al. [3].

1.3.3 Diverse palindromic factorization

We define a restricted variant of palindromic factorization, called the *diverse palindromic factorization*. A factorization of a string w is said to be *diverse* if the factors are distinct from each other. Some well-known factorizations, such as the LZ78 [79] factorization are diverse (except that the last factor may have appeared before). Fernau et al. [30] recently proved that it is NP-complete to determine whether a given string has a diverse factorization of at least a given size, and Schmid [68] has investigated related questions. It seems natural to consider the problem of determining whether a given string has a diverse factorization into palindromes. For example, `abcddeef` and `abcdefed` each have exactly one such factorization — i.e., `a`, `b`, `c`, `dd`, `ee`, `f` and `a`, `b`, `c`, `defed`, respectively — but `abcdefdc` has none. This problem is obviously in NP and we prove that it is NP-hard and, thus, NP-complete by showing a reduction from the circuit satisfiability problem [58].

We also show that it is NP-complete for any fixed k to decide whether a given string can be factored into palindromes that each appear at most k times in the factorization; we call such

a factorization k-*diverse*. Finally, since several recent papers (e.g. [15, 16, 40]) consider the effect of alphabet size on the difficulty of various string problems, we show that the problems remain NP-complete even if the string is restricted to be binary.

1.3.4 Closed factorization

A *closed string* is a string with a proper substring that occurs as a prefix and a suffix but does not have internal occurrences. Closed strings were introduced by Fici [31] as objects of combinatorial interest in the study of Trapezoidal and Sturmian words. Since then, Badkobeh, Fici, and Liptak [9] have proved a tight lower bound for the number of closed substrings in strings of given length and alphabet.

In this thesis, we initiate the study of algorithms for computing closed factors. In particular we consider two algorithmic problems. The first, which we call the *closed factorization problem*, is to greedily factorize a given string into a sequence of longest closed factors (we give a formal definition of the problem in Section 6.1). We describe an algorithm for this problem that uses $O(n)$ time and space, where n is the length of the given string.

The second problem we consider is the *closed factor array problem*, which requires us to compute the length of the longest closed factor starting at each position in the input string. We show that this problem can be solved in $O(n\frac{\log n}{\log\log n})$ time, using techniques from computational geometry.

1.3.5 Abelian regularities

Two strings s_1 and s_2 are said to be *Abelian equivalent* if s_1 is a permutation of s_2, or vice versa. For instance, strings `ababaac` and `caaabba` are Abelian equivalent. Since the seminal paper by Erdős [29] published in 1961, the study of Abelian equivalence on strings has attracted much attention, both in word combinatorics and string algorithmics.

We are interested in the following algorithmic problems related to Abelian regularities of strings.

1. Compute *Abelian squares* in a given string.
2. Compute *regular Abelian periods* of a given string.
3. Compute *longest common Abelian factors* of two given strings.

Cummings and Smyth [24] proposed an $O(n^2)$-time algorithm to solve Problem 1, where n is the length of the given string. Crochemore et al. [20] proposed an alternative $O(n^2)$-time solution to the same problem. Recently, Kociumaka et al. [51] showed how to compute all Abelian squares in $O(s + \frac{n^2}{\log^2 n})$ time, where s is the number of outputs.

Related to Problem 2, various kinds of Abelian periods of strings have been considered: An integer p is said to be a *full Abelian period* of a string w iff there is a decomposition $u_1, \ldots, u_z$ of w such that $|u_i| = p$ for all $1 \leq i \leq z$ and $u_1, \ldots, u_z$ are all Abelian equivalent. A pair (p, t) of integers is said to be a *regular Abelian period* (or simply an *Abelian period*) of a string w iff there is a decomposition $v_1, \ldots, v_s$ of w such that p is a full Abelian period of $v_1 \cdots v_{s-1}$, $|v_i| = p$ for all $1 \leq i \leq s-1$, and v_s is a permutation of a substring of v_1 (and hence $t \leq p$). A triple (h, p, t) of integers is said to be a *weak Abelian period* of a string w iff there is a decomposition $y_1, \ldots, y_r$ of w such that (p, t) is an Abelian period of $y_2 \cdots y_r$, $|y_1| = h$, $|y_i| = p$ for all $2 \leq i \leq r-1$, $|y_r| = t$, and y_1 is a permutation of a substring of y_2 (and hence $h \leq p$).

The study on Abelian periodicity of strings was initiated by Constantinescu and Ilie [18]. Fici et al. [34] gave an $O(n \log \log n)$-time algorithm to compute all full Abelian periods. Later, Kociumaka et al. [50] showed an optimal $O(n)$-time algorithm to compute all full Abelian periods. Fici et al. [34] also showed an $O(n^2)$-time algorithm to compute all regular Abelian periods for a given string of length n. Kociumaka et al. [50] also developed an algorithm which finds all regular Abelian periods in $O(n(\log \log n + \log \sigma))$ time, where σ is the alphabet size. Fici et al. [33] proposed an algorithm which computes all weak Abelian periods in $O(\sigma n^2)$ time, and later Crochemore et al. [20] proposed an improved $O(n^2)$-time algorithm to compute all weak Abelian periods. Kociumaka et al. [51] showed how to compute all *shortest* weak Abelian periods in $O(n^2/\sqrt{\log n})$ time.

Consider two strings w_1 and w_2. A pair (s_1, s_2) of a substring s_1 of w_1 and a substring s_2 of w_2 is said to be a *common Abelian factor* of w_1 and w_2, iff s_1 and s_2 are Abelian equivalent. Alatabbi et al. [2] proposed an $O(\sigma n^2)$-time and $O(\sigma n)$-space algorithm to solve Problem 3 of computing all *longest common Abelian factors* (*LCAFs*) of two given strings of total length n. Later, Grabowski [39] showed an algorithm which finds all LCAFs in $O(\sigma n^2)$ time with $O(n)$ space. He also presented an $O((\frac{\sigma}{k} + \log \sigma) n^2 \log n)$-time $O(kn)$-space algorithm for a parameter $k \leq \frac{\sigma}{\log \sigma}$. Recently, Badkobeh et al. [10] proposed an $O(n \log^2 n \log^* n)$-time $O(n \log^2 n)$-space algorithm for finding all LCAFs.

In this thesis, we show that we can accelerate computation of Abelian regularities of strings

via *run length encoding* (*RLE*) of strings. Namely, if m is the size of the RLE of a given string w of length n, we show that:

1. All Abelian squares in w can be computed in $O(mn)$ time.
2. All regular Abelian periods of w can be computed in $O(mn)$ time.
3. All longest common Abelian factors of w_1 and w_2 can be computed in $O(m^2n)$ time.

Since $m \leq n$ always holds, our $O(mn)$-time solution to Problem 1 is at least as efficient as the $O(n^2)$-time solutions by Cummings and Smyth [24] and by Crochemore et al. [20], and can be much faster when the input string w is highly compressible by RLE. Amir et al. [5] proposed an $O(\sigma(m^2 + n))$-time algorithm to compute all Abelian squares using RLEs. Our $O(mn)$-time solution is faster than theirs when $\frac{\sigma m^2}{m-\sigma} = \omega(n)$.

Our $O(mn)$-time solution to Problem 2 is faster than the $O(n(\log\log n + \log\sigma))$-time solution by Kociumaka et al. [50] for highly RLE-compressible strings with $\log\log n = \omega(m)$[2].

Our $O(m^2n)$-time solution to Problem 3 is faster than the $O(\sigma n^2)$-time solution by Grabowski [39] when $\sigma n = \omega(m^2)$, and is faster than the fastest variant of the other solution by Grabowski [39] (choosing $k = \frac{\sigma}{\log\sigma}$) when $\sqrt{n\log n\log\sigma} = \omega(m)$. Also, our solution is faster than the $O(n\log^2 n\log^* n)$-time solution by Badkobeh et al. [10] when $\log n\sqrt{\log^* n} = \omega(m)$. The time bounds of our algorithms are all deterministic.

1.4 Organization

The rest of this thesis is organized as follows: In Chapter 3, we introduce the reversed LZ factorization without self-references, and consider computing the reversed LZ factorization with or without self-references of a given string w in a small space. In Chapter 4, we define the problems of computing a smallest-sized palindromic factorization of w and of computing a smallest-sized palindromic cover of w, and present efficient algorithms for them. In Chapter 5, we formulate the diverse palindromic factorization problem and prove that the existence problem of the diverse palindromic factorization is NP-complete. In Chapter 6, we define the closed factorization problem and the longest closed factor array problem, and show algorithms for solving them. In Chapter 7, we consider accelerating computation of various Abelian regularities Abelian squares: Abelian periods, and longest common Abelian factors via Run Length factorization.

[2]Since we can w.l.o.g. assume that $\sigma \leq m$, the $\log\sigma$ term is negligible here.

Chapter 2

Preliminaries

2.1 Notion and notation

Let Σ be an alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are called a *prefix*, a *substring*, and a *suffix* of the string $w = xyz$, respectively. A prefix, substring, and suffix of a string w is said to be *proper* if it is not w. A string b is called a *border* of another string w if b is both a proper prefix and suffix of w. The sets of substrings and suffixes of w are denoted by $Substr(w)$ and $Suffix(w)$, respectively. The *length* of string w is denoted by $|w|$. The empty string ε is a string of length 0, that is, $|\varepsilon| = 0$. For $1 \leq i \leq |w|$, $w[i]$ denotes the i-th character of w. For $1 \leq i \leq j \leq |w|$, $w[i..j]$ denotes the substring of w that begins at position i and ends at position j. For convenience, let $w[i..j] = \varepsilon$ for $i > j$.

A *period* of a string x is an integer p with $0 < p \leq |x|$ such that $x[i] = x[i+p]$ for all i with $1 \leq i \leq |x| - p$.

Proposition 1 (Periodicity Lemma [23])**.** *Let d and d' be two periods of a string w. If $d + d' - \gcd(d, d') \leq |w|$, then $\gcd(d, d')$ is also a period of w.*

Let w^{rev} denote the reversed string of s, that is, $w^{\mathrm{rev}} = w[|w|] \cdots w[2]w[1]$. A string x is called a *palindrome* if $x = x^{\mathrm{rev}}$. The *center* of a palindromic substring $w[i..j]$ of a string w is $\frac{i+j}{2}$. The *radius* of palindrome x is $\frac{|x|}{2}$. A palindromic substring $w[i..j]$ is called the *maximal palindrome* at the center $\frac{i+j}{2}$ if no other palindromes at the center $\frac{i+j}{2}$ have a larger radius than $w[i..j]$, i.e., if $w[i-1] \neq w[j+1]$, $i = 1$, or $j = |w|$. Since a string w of length $n \geq 1$ has $2n - 1$ centers $(1, 1.5, \ldots, n - 0.5, n)$, w has exactly $2n - 1$ maximal palindromes. Note that every palindromic suffix of w is a maximal palindrome of w.

For any string w of length $n \geq 1$, let $Spals(w)$ denote the set of the beginning positions of

the palindromic suffixes of w, i.e.,

$$Spals(w) = \{n - |s| + 1 \mid s \in Suffix(w), s \text{ is a palindrome}\}.$$

We introduce the property of $Spals$.

Proposition 2 ([6, 62])**.** *For any string w of length n, $Spals(w)$ can be represented by $O(\log n)$ arithmetic progressions.*

A *factorization* of a string w is a sequence $f_1, \ldots, f_k$ of non-empty strings such that $w = f_1 \cdots f_k$. Each f_i is called a *factor* of the factorization. The *size* of the factorization is the number k of factors in it.

2.2 Model of computation

In this thesis, we use the unit cost word RAM model[1], where we have access to a random access memory consisting of cells, each of which stores an ℓ-bit word, and all the basic arithmetic and logic operations over ℓ-bit words can be carried out in constant time. Let n be the size of data in memory. In this model it is usually assumed that $\ell = \Omega(\log n)$, or simply $\ell \geq \lceil \log n \rceil$, which means that with one machine word we can address any data element.

There are three types of alphabets from which symbols are drawn: (i) a constant-sized alphabet, (ii) an integer alphabet where symbols are integers in the range $[1..n^c]$ for a constant c, and (iii) a general alphabet in which the only operations on strings are symbols comparisons.

2.3 Tools

2.3.1 Suffix array

The suffix array [61] SA_w (we drop subscripts when they are clear from the context) of a string w is an array $SA[1..n]$ which contains a permutation of the integers $[1..n]$ such that $w[SA[1]..n] < w[SA[2]..n] < \cdots < w[SA[n]..n]$. In other words, $SA[j] = i$ iff $w[i..n]$ is the j^{th} suffix of w in ascending lexicographical order.

[1] RAM stands for Random Access Machine.

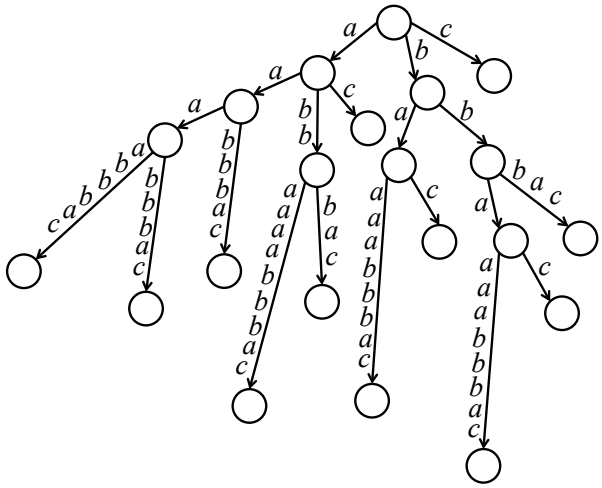

Figure 2.1: $STree(w)$ with $w = abbaaaabbbac$.

2.3.2 Suffix tree

The suffix tree [74] of string s, denoted $STree(s)$, is a rooted tree such that

1. Each edge is labeled with a non-empty substring of s, and each path from the root to a node spells out a substring of s;

2. Each internal node v has at least two children, and the labels of distinct out-going edges of v begin with distinct characters;

3. For each suffix x of w, there is a path from the root that spells out x.

The number of nodes and edges of $STree(s)$ is $O(|s|)$, and $STree(s)$ can be represented using $O(|s| \log |s|)$ bits of space, by implementing each edge label y as a pair (i, j) such that $y = s[i..j]$.

For a constant alphabet, Weiner's algorithm [74] constructs $STree(s^{\text{rev}})$ in an online manner from left to right, i.e., constructs $STree(s[1..j]^{\text{rev}})$ in increasing order of $j = 1, 2, \ldots, |s|$, in $O(|s|)$ time using $O(|s| \log |s|)$ bits of space. It is known that the tree of the suffix links of the directed acyclic word graph [12] of s forms $STree(s^{\text{rev}})$. Hence, for larger alphabets, we have the following:

Proposition 3 ([12])**.** *Given a string s, we can compute $STree(s^{\text{rev}})$ online from left to right, in $O(|s| \log \sigma)$ time using $O(|s| \log |s|)$ bits of space.*

2.3.3 Suffix trie

In our algorithms, we will also use the generalized suffix *trie* for a set W of strings, denoted $STrie(W)$. $STrie(W)$ is a rooted tree such that

1. Each edge is labeled with a character, and each path from the root to a node spells out a substring of some string $w \in W$;
2. The labels of distinct out-going edges of each node must be different;
3. For each suffix s of each string $w \in W$, there is a path from the root that spells out s.

2.3.4 Knuth-Morris-Pratt algorithm

Let P be a non-empty string called the pattern. The Knuth-Morris-Pratt (KMP for short) algorithm [49] is based on the KMP automaton, which accepts the language $\Sigma^* P$. Let $Q = \{0, 1, \ldots, |P|\}$ be the set of states, and let *fail* be a special value not in Q. The state-transition $\delta : Q \times \Sigma \to Q$ of the KMP automaton for P is represented as the two functions: the goto function $g : Q \times \Sigma \to Q \cup \{\textit{fail}\}$, and the failure function $f : Q - \{0\} \to Q$ satisfying

$$\delta(j, a) = \begin{cases} g(j, a), & \text{if } g(j, a) \neq \textit{fail}; \\ \delta(f(j), a), & \text{otherwise.} \end{cases}$$

The goto function g takes $j \in Q$ and $a \in \Sigma$ as input and returns $j + 1$ if $P[j + 1] = a$, otherwise, returns *fail*. (The case $j = 0$ is an exception. Let $g(0, a) = 0$ for every $a \in \Sigma$ with $P[1] \neq a$.) The failure function f takes $j \in Q - \{0\}$ as input and returns the length of longest border of $P[1..j]$. Figure 2.2 shows the KMP automaton for $P = abacb$ with $\Sigma = \{a, b, c\}$. The move of the KMP automaton of Figure 2.2 on text $T = abacbbaababacbb$ is shown in Figure 2.3.

Proposition 4. *The KMP algorithm finds all occurrences of a pattern P of length m within a text T of length n in $O(m + n)$ time using $O(m)$ space for a general unordered alphabet.*

2.4 Two variants of LZ77 factorization

We give formal definitions of two variants of LZ77 factorization as below.

Definition 1 (LZ77 factorization without self-references)**.** *The LZ77 factorization without self-reference of a string w is a factorization $f_1, \ldots, f_m$ such that for any $1 \leq i \leq m$,*

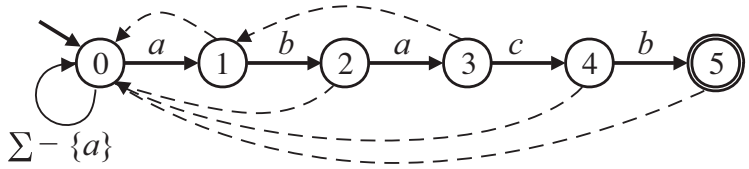

Figure 2.2: KMP automaton for $P = abacb$ is displayed. The circles denote the states, and the thick circle means the final state. The solid and the broken arrows represent the goto and the failure functions, respectively.

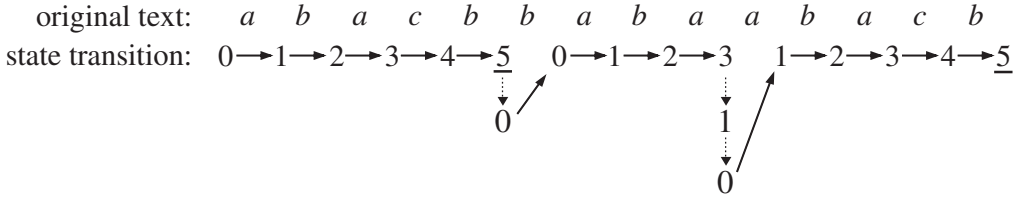

Figure 2.3: Move of KMP automaton is demonstrated. The solid and the broken arrows represent the state transitions with the goto and the failure functions, respectively. The underlined number indicates that the pattern occurs.

> *f_i is the longest non-empty prefix of $w[j..|w|]$ that occurs in $w[1..j-1]$ if such exists, and $f_i = w[j]$ otherwise,*

where $j = |f_1 \cdots f_{i-1}| + 1$.

Definition 2 (LZ77 factorization with self-references)**.** *The LZ77 factorization with self-reference of a string w is a factorization $f_1, \ldots, f_m$ such that for any $1 \leq i \leq m$,*

> *f_i is the longest non-empty prefix of $w[j..|w|]$ that occurs in $w[1..|f_1 \cdots f_i| - 1]$ if such exists, and $f_i = w[j]$ otherwise,*

where $j = |f_1 \cdots f_{i-1}| + 1$.

Chapter 3

Reversed LZ Factorization Online

In this chapter, we address two variants of the LZ77 factorization, called the reversed LZ factorization with and without self-references, and present efficient online algorithms for computing them.

The results in this chapter were originally published in [71].

3.1 Notation

For an input string w of length n over an alphabet of size $\sigma \leq n$, let $r = \frac{\log_\sigma n}{4} = \frac{\log n}{4 \log \sigma}$. For simplicity, assume that $\log n$ is divisible by $4 \log \sigma$, and that n is divisible by r. A string of length r, called a *meta-character*, fits in a single machine word. Thus, a meta-character can also be transparently regarded as an element in the integer alphabet $\Sigma^r = \{1, \ldots, n\}$. We assume that given $1 \leq i \leq n-r+1$, any meta-character $A = w[i..i+r-1]$ can be retrieved in constant time.

We call a string on the alphabet Σ^r of meta-characters, a *meta-string*. Any string w whose length is divisible by r can be viewed as a meta-string w of length $m = \frac{n}{r}$. We write $\langle w \rangle$ when we explicitly view string w as a meta-string, where $\langle w \rangle[j] = w[(j-1)r + 1..jr]$ for each $j \in [1, m]$. Such range $[(j-1)r + 1, jr]$ of positions will be called *meta-blocks* and the beginning positions $(j-1)r + 1$ of meta-blocks will be called *block borders*. For clarity, the length m of a meta-string $\langle w \rangle$ will be denoted by $\|\langle w \rangle\|$. Note that $m \log n = n \log \sigma$.

3.2 Reversed LZ factorization

Kolpakov and Kucherov [53] introduced the following variant of LZ77 factorization.

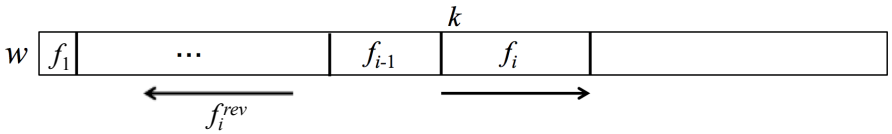

Figure 3.1: Let $k = |f_1 \cdots f_{i-1}| + 1$. f_i is the longest non-empty prefix of $w[k..|w|]$ that is also a substring of $w[1..k-1]^{\text{rev}}$ if such exists.

Definition 3 (Reversed LZ factorization without self-references)**.** *The reversed LZ factorization of a string w without self-references, denoted $RLZ(w)$, is a factorization $f_1, \ldots, f_m$ of w such that for any $1 \leq i \leq m$,*

> *f_i is the longest non-empty prefix of $w[k..|w|]$ that occurs in $w[1..k-1]^{\text{rev}}$ if such exists, and $f_i = w[k]$ otherwise,*

where $k = |f_1 \cdots f_{i-1}| + 1$.

Figure 3.1 illustrates Definition 3.

Example 1. *For string $w = abbaaaabbbac$, $RLZ(w)$ consists of the following factors: $f_1 = a$, $f_2 = b$, $f_3 = ba$, $f_4 = a$, $f_5 = aabb$, $f_6 = ba$, and $f_7 = c$.*

We are interested in online computation of $RLZ(w)$. Using Proposition 3, one can compute $RLZ(w)$ online in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space [53], where $n = |w|$. The idea is as follows: Assume we have already computed the first j factors $f_1, f_2, \ldots, f_j$, and we have constructed $STree(w[1..l_j]^{\text{rev}})$, where $l_j = \sum_{h=1}^{j} |f_h|$. Now the next factor f_{j+1} is the longest prefix of $w[l_j + 1..n]$ that is represented by a path from the root of $STree(w[1..l_j]^{\text{rev}})$. After the computation of f_{j+1}, we update $STree(w[1..l_j]^{\text{rev}})$ to $STree(w[1..l_{j+1}]^{\text{rev}})$, using Proposition 3. In the next section, we will propose a new space-efficient online algorithm which requires $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space.

We introduce yet another new variant, the reversed LZ factorization *with* self-references.

Definition 4 (Reversed LZ factorization with self-references)**.** *The reversed LZ factorization of a string w with self-references, denoted $RLZS(w)$, is a factorization $g_1, \ldots, g_p$ of w such that for any $1 \leq i \leq p$,*

> *g_i is the longest non-empty prefix of $w[k..|w|]$ that occurs in $w[1..|g_1 \cdots g_i| - 1]^{\text{rev}}$ if such exists, and $g_i = w[k]$ otherwise,*

where $k = |g_1 \cdots g_{i-1}| + 1$.

Figure 3.2 illustrates Definition 4.

Example 2. *For string* $w = abbaaaabbbac$, $RLZS(w)$ *consists of the following factors:* $g_1 = a$, $g_2 = b$, $g_3 = baaaabb$, $g_4 = ba$, *and* $g_5 = c$.

Note that in Definition 4 the ending position of a previous occurrence of g_i^{rev} does not have to be prior to the beginning position k of g_i, while in Definition 3 it has to. This is the difference between $RLZ(w)$ and $RLZS(w)$.

In this chapter we propose two online algorithms to compute $RLZS(w)$; the first one runs in $O(n \log \sigma)$ time using $O(n \log n)$ bits of space, and the second one does in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space.

3.3 Computing $RLZ(w)$ in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space

The outline of our online algorithm to compute $RLZ(w)$ follows the algorithm of Starikovskaya [70] which computes LZ 77 factorization [78] in an online manner and in $O(n \log^2 n)$ time using $O(n \log \sigma)$ bits of space. The Starikovskaya algorithm maintains the suffix tree of the meta-string $\langle w \rangle$ in an online manner, i.e., maintains $STree(\langle w \rangle[1..k])$ in increasing order of $k = 1, 2, \ldots, n/r$, and maintains a generalized suffix trie for a set of substrings of $w[1..kr]$ of length $2r$ that begin at a block border. In contrast to the Starikovskaya algorithm, our algorithm maintains $STree((\langle w \rangle[1..k])^{\mathrm{rev}})$ in increasing order of $k = 1, 2, \ldots, n/r$, and maintain a generalized suffix trie for a set of substrings of $w[1..kr]^{\mathrm{rev}}$ of length $2r$ that begin at a block border.

Assume we have already computed the first $i - 1$ factors $f_1, \ldots, f_{i-1}$ of $RLZ(w)$ and are computing the ith factor f_i. Let $l_i = \sum_{j=1}^{i-1} |f_j|$. This implies that we have processed

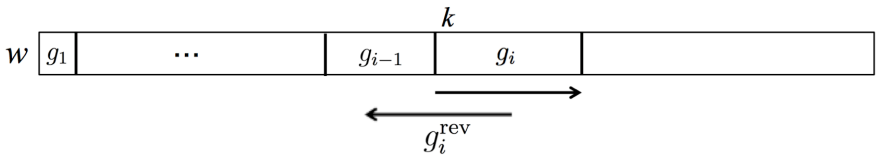

Figure 3.2: Let $k = |g_1 \cdots g_{i-1}| + 1$. g_i is the longest non-empty prefix of $w[k..|w|]$ that is also a substring of $w[1..|g_1 \cdots g_i| - 1]^{\mathrm{rev}}$ if such exists.

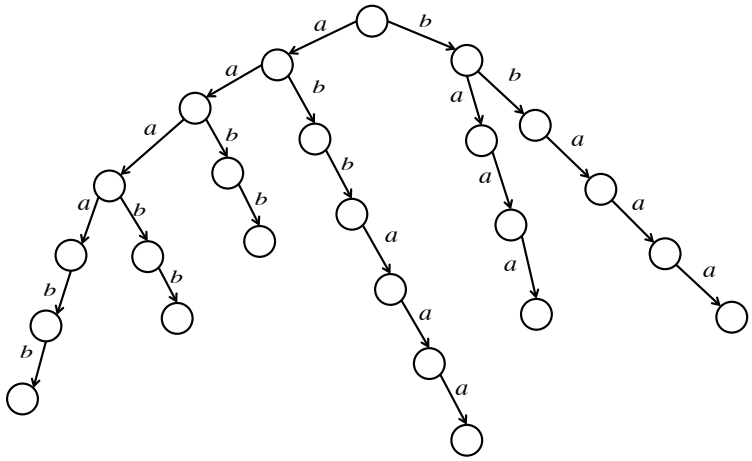

Figure 3.3: Let $r = 3$ and consider string $w = bba|aaa|bba|bac$, where $|$ represents a block border. The figure shows $STrie(W_3^{\text{rev}})$ where $W_3^{\text{rev}} = \{aaaabb, abbaaa\}$.

$(\langle w \rangle[1..k])^{\text{rev}}$ where $k = \lceil l_i/r \rceil$, i.e., the kth meta block contains position l_i. As is the case with the Starikovskaya algorithm, our algorithm consists of two main phrases, depending on whether $|f_i| < r$ or $|f_i| \geq r$.

3.3.1 Algorithm for $|f_i| < r$

For any k ($1 \leq k \leq n/r$), let W_k^{rev} denote the set of substrings of $w[1..kr]^{\text{rev}}$ of length $2r$ that begin at a block border, i.e., $W_k^{\text{rev}} = \{w[tr + 1..(t + 2)r]^{\text{rev}} \mid 1 \leq t \leq (k - 2)\}$. We maintain $STrie(W_k^{\text{rev}})$ in an online manner, for $k = 1, 2, \ldots, n/r$. Note that $STrie(W_k^{\text{rev}})$ represents all substrings of $w[1..kr]^{\text{rev}}$ of length r which do not necessarily begin at a block border. Therefore, we can use $STrie(W_k^{\text{rev}})$ to determine if $|f_i| < r$, and if so, compute f_i. An example for $STrie(W_k^{\text{rev}})$ is shown in Figure 3.3.

A minor issue is that $STrie(W_k^{\text{rev}})$ may contain "unwanted" substrings that do not correspond to a previous occurrence of f_i^{rev} in $w[1..l_i]$, since substrings $w[(k - 2)r + 1..y]^{\text{rev}}$ for any $l_i < y \leq kr$ are represented by $STrie(W_k^{\text{rev}})$. In order to avoid finding such unwanted occurrences of f_i^{rev}, we associate to each node v representing a reversed substring x^{rev}, the leftmost ending position of x in $w[1..kr]$. Assume we have traversed the prefix of length $p \geq 0$ of $w[l_i + 1..n]$ in the trie, and all the nodes involved in the traversal have positions smaller than $l_i + 1$. If either the node representing $w[l_i + 1..l_i + p + 1]$ stores a position larger than l_i or there is no node representing $w[l_i + 1..l_i + p + 1]$, then $f_i = w[l_i + 1..l_i + p]$ if $p \geq 1$, and $f_i = w[l_i + 1]$ if $p = 0$.

As is described above, f_i can be computed in $O(|f_i| \log \sigma)$ time. When $l_i + p > kr$, we insert the suffixes of a new substring $w[(k-1)r+1..(k+1)r]^{\mathrm{rev}}$ of length $2r$ into the trie, and obtain the updated trie $STrie(W_{k+1}^{\mathrm{rev}})$. Since there exist $\sigma^{2r} = \sigma^{\frac{\log n}{2}} = \sqrt{n}$ distinct strings of length $2r$, the number of nodes in the trie is bounded by $O(\sqrt{n}r^2) = O(\sqrt{n}(\log_\sigma n)^2)$. Hence the trie requires $o(n)$ bits of space. Each update adds $O(r^2)$ new nodes and edges into the trie, taking $O(r^2 \log \sigma)$ time. Since there are n/r blocks, the total time complexity to maintain the trie is $O(nr \log \sigma) = O(n \log n)$.

The above discussion leads to the following lemma:

Lemma 1. *We can maintain in $O(n \log n)$ total time, a dynamic data structure occupying $o(n)$ bits of space that allows whether or not $|f_i| < r$ to be determined in $O(|f_i| \log \sigma)$ time, and if so, computes f_i and a previous occurrence of f_i^{rev} in $O(|f_i| \log \sigma)$ time.*

3.3.2 Algorithm for $|f_i| \geq r$

Assume we have found that the length of the longest prefix of $w[l_i+1..n]$ that is represented by $STrie(W_k^{\mathrm{rev}})$ is at least r, which implies that $|f_i| \geq r$.

For any string f and integer $0 \leq m \leq \min(|f|, r-1)$, let strings $\alpha_m(f)$, $\beta_m(f)$, $\gamma_m(f)$ satisfy $f = \alpha_m(f)\beta_m(f)\gamma_m(f)$, $|\alpha_m(f)| = m$, and $|\beta_m(f)| = j'r$ where $j' = \max\{j \geq 0 \mid m + jr \leq |f|\}$. We say that an occurrence of f in w has offset m ($0 \leq m \leq r-1$), if, in the occurrence, $\alpha_m(f)$ corresponds to a suffix of a meta-block, $\beta_m(f)$ corresponds to a sequence of meta-blocks (i.e. $\beta_m(f) \in Substr(\langle w \rangle)$), and $\gamma_m(f)$ corresponds to a prefix of a meta-block. Let f_i^m denote the longest prefix of $w[l_i+1..n]$ which has a previous occurrence in $w[1..l_i]$ with offset m. Thus, $|f_i| = \max_{0 \leq m < r} |f_i^m|$.

Our algorithm maintains two suffix trees on meta-strings, $STree((\langle w \rangle[1..k-1])^{\mathrm{rev}})$ and $STree((\langle w \rangle[1..k])^{\mathrm{rev}})$. Depending on the value of m, we use either $STree((\langle w \rangle[1..k-1])^{\mathrm{rev}})$ and $STree((\langle w \rangle[1..k])^{\mathrm{rev}})$.

If $l_i - (k-1)r \geq m$, i.e. the distance between the $(k-1)$th block border and position l_i is not less than m, then we use $STree((\langle w \rangle[1..k])^{\mathrm{rev}})$ to find f_i^m. We associate to each internal node v of $STree((\langle w \rangle[1..k])^{\mathrm{rev}})$ the lexicographical ranks of the leftmost and rightmost leaves in the subtree rooted at v, denoted $left(v)$ and $right(v)$, respectively. Recall that the leaves of $STree((\langle w \rangle[1..k])^{\mathrm{rev}})$ correspond to the block borders $1, r+1, \ldots, (k-1)r+1$. Hence, $\alpha_m(f_i^m)\beta_m(f_i^m)$ occurs in $w[1..l_i]^{\mathrm{rev}}$ iff there is a node v representing $\beta_m(f_i^m)$ and the interval $[left(v), right(v)]$ contains at least one block border b such that $w[b-m..b-1] = \alpha_m(f_i^m)$. To

determine $\gamma_m(f_i^m)$, at each node v of $STree((\langle w \rangle[1..k])^{\text{rev}})$ we maintain a trie T_v that stores the first meta-characters of the outgoing edge labels of v. Then, $\alpha_m(f_i^m)\beta_m(f_i^m)\gamma_m(f_i^m)$ occurs in $w[1..l_i]^{\text{rev}}$ iff there is a node u of T_v representing $\gamma_m(f_i^m)$ and the interval $[\mathit{left}(u_1), \mathit{right}(u_2)]$ contains at least one block border b such that $w[b-m..b-1] = \alpha_m(f_i^m)$, where u_1 and u_2 are respectively the leftmost and rightmost children of u in T_v.

If $l_i - (k-1)r < m$, i.e. if the the distance between the $(k-1)$th block border and position l_i is less than m, then we use $STree((\langle w \rangle[1..k-1])^{\text{rev}})$ to find f_i^m. This allows us to find only previous occurrences of f_i^{rev} that end before $l_i + 1$. All the other procedures follow the case where $l_i - (k-1)r \geq m$, mentioned above.

Lemma 2. *We can maintain in $O(n \log^2 n)$ total time, a dynamic data structure occupying $O(n \log \sigma)$ bits of space that allows to compute f_i with $|f_i| \geq r$ and a previous occurrence of f_i^{rev} in $O(|f_i| \log^2 n)$ time.*

Proof. Traversing the suffix tree for $\beta_m(f_i^m)$ takes $O(\frac{|f_i^m|}{r} \log n) = O(|f_i^m| \log \sigma)$ time since $\|\langle \beta_m(f_i^m) \rangle\| \leq |\frac{f_i^m}{r}|$. Also, traversing the trie for $\gamma_m(f_i^m)$ takes $O(r \log \sigma)$ time, since $|\gamma_m(f_i^m)| < r$. To assure $\beta_m(f_i^m)\gamma_m(f_i^m)$ is immediately preceded by $\alpha_m(f_i^m)$, we use the dynamic data structure proposed by Starikovskaya [70] which is based on the dynamic wavelet trees [59]. At each node v, the data structure allows us to check if the interval $[\mathit{left}(v), \mathit{right}(v)]$ contains a block border of interest in $O(\log^2 n)$ time, and to insert a new element to the data structure in $O(\log^2 n)$ time. Thus, f_i can be computed in $O(\sum_{0 \leq m \leq r-1}(|f_i^m| \log \sigma + r \log \sigma + |\frac{f_i^m}{r}| \log^2 n)) = O(|f_i| \log^2 n)$. The position of a previous occurrence of f_i^{rev} can be retrieved in constant time, since each leaf of the suffix tree corresponds to a block border. Once f_i is computed, we update $STree((\langle w \rangle[1..k])^{\text{rev}})$ to $STree((\langle w \rangle[1..k'])^{\text{rev}})$, such that the k'th block border contains position l_{i+1} in w. Using Proposition 3, the suffix tree can be maintained in a total of $O(\frac{n}{r} \log \sigma) = O(n \log n)$ time.

It follows from Proposition 3 that the suffix tree on meta-strings requires $O(\frac{n}{r} \log n) = O(n \log \sigma)$ bits of space. Since the dynamic data structure of Starikovskaya [70] takes $O(n \log \sigma)$ bits of space, the total space complexity of our algorithm is $O(n \log \sigma)$ bits. □

The main result of this section follows from Lemma 1 and Lemma 2:

Theorem 1. *Given a string w of length n, we can compute $RLZ(w)$ in an online manner, in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space.*

3.4 Online computation of reversed LZ factorization with self-references

In this section, we consider to compute $RLZS(w)$ for a given string w in an online manner. An interesting property of the reversed LZ factorization with self-references is that, the factorization can significantly change when a new character is appended to the end of the string. A concrete example is shown in Figure 3.4, which illustrates online computation of $RLZS(w)$ with $w = abbaaaabbbac$. Focus on the factorization of $abbaaaab$. Although there is a factor starting at position 5 in $RLZS(abbaaaab)$, there is no factor starting at position 5 in $RLZS(abbaaaabb)$. Below, we will characterize this with its close relationship to palindromes.

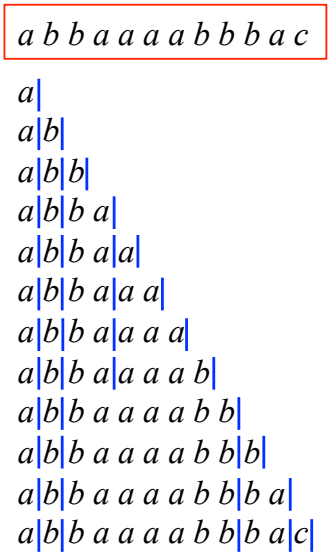

Figure 3.4: A snapshot of online computation of $RLZS(w)$ with $w = abbaaaabbbac$. For each non-empty prefix $w[1..k]$ of w, | denotes the boundary of factors in $RLZS(w[1..k])$.

3.4.1 Computing $RLZS(w)$ in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space

Let w be any string of length n. For any $1 \leq j \leq n$, the occurrence of substring p starting at position j is called self-referencing, if there exists j' such that $w[j'..j' + |p| - 1]^{\mathrm{rev}} = w[j..j + |p| - 1]$ and $j \leq j' + |p| - 1 < j + |p| - 1$.

For any $1 \leq k \leq n$, let $Lpal_w(k) = \max\{k - j + 1 \mid w[j..k] = w[j..k]^{\mathrm{rev}}, 1 \leq j \leq k\}$. That is, $Lpal_w(k)$ is the length of the longest palindrome that ends at position k in w.

Lemma 3. *For any string w of length n and $1 \leq k \leq n$, let $RLZS(w[1..k-1]) = g_1, \ldots, g_p$. Let $l_q = \sum_{h=1}^{q} |g_h|$ for any $1 \leq q \leq p$. Then*

$$RLZS(w[1..k]) = \begin{cases} g_1, \ldots, g_p w[k] & \text{if } g_p w[k] \in Substr(w[1..l_{p-1}]^{\mathrm{rev}}) \text{ and } l_{p-1}+1 \leq d_k, \\ g_1, \ldots, g_p, w[k] & \text{if } g_p w[k] \notin Substr(w[1..l_{p-1}]^{\mathrm{rev}}) \text{ and } l_{p-1}+1 \leq d_k, \\ g_1, \ldots, g_j, w[l_j+1..k] & \text{otherwise}, \end{cases}$$

where $d_k = k - Lpal_w(k) + 1$ and j is the minimum integer such that $l_j \geq d_k$.

Proof. By definition of $Lpal_w(k)$ and d_k, $w[d_k..k]$ is the longest suffix palindrome of $w[1..k]$. If $l_{p-1}+1 \leq d_k$, $w[l_{p-1}+1..k]$ cannot be self-referencing. Hence the first and the second cases of the lemma follow. Consider the third case. Since $l_j \geq d_k$, $w[l_j+1..k]$ is self-referencing. Since $RLZS(w[1..l_j]) = g_1, \ldots, g_j$, the third case follows. □

See Figure 3.4 and focus on $RLZS(abbaaaab)$, where $g_1 = a$, $g_2 = b$, $g_3 = ba$, and $g_4 = aaab$. Consider to compute $RLZS(abbaaaabb)$. Since the longest suffix palindrome $bbaaaabb$ intersects the boundary between g_3 and g_4 of $RLZS(abbaaaab)$, the third case of Lemma 3 applies. Consequently, the new factorization $RLZS(abbaaaabb)$ consists of $g_1 = a$ and $g_2 = b$ of $RLZS(abbaaaab)$, and a new self-referencing factor $g_3 = baaaabb$.

Theorem 2. *Given a string w of length n, we can compute $RLZS(w)$ in an online manner, in $O(n \log \sigma)$ time and $O(n \log n)$ bits of space.*

Proof. Suppose we have already computed $RLZS(w[1..k-1])$, and we are computing $RLZS(w[1..k])$ for $1 \leq k \leq n$.

Assume $l_{p-1}+1 \leq d_k$. We check whether $g_p w[k] \in Substr(w[1..l_{p-1}]^{\mathrm{rev}})$ or not using $STree(w[1..l_{p-1}]^{\mathrm{rev}})$. If the first case of Lemma 3 applies, then we proceed to the next position $k+1$ and continue to traverse the suffix tree. If the second case of Lemma 3 applies, then we update the suffix tree for the reversed string, and proceed to computing $RLZS(w[1..k+1])$.

Assume $l_{p-1}+1 > d_k$, i.e., the third case of Lemma 3 holds. For every $j < e \leq p$, we remove g_e of $RLZS(w[1..k-1])$, and the last factor of $RLZS(w[1..k])$ is $w[l_j+1..k]$. We then proceed to computing $RLZS(w[1..k+1])$.

As is mentioned in Section 3.2, in a total of $O(n \log \sigma)$ time and $O(n \log n)$ bits of space, we can check whether the first or the second case of Lemma 3 holds, as well as maintain the

suffix tree for the reversed string online. In order to compute $Lpal_w(k)$ in an online manner, we can use Manacher's algorithm [60] which computes the maximal palindromes for all centers in w in $O(n)$ time and in an online manner. Since Manacher's algorithm actually maintains the center of the longest suffix palindrome of $w[1..k]$ when processing $w[1..k]$, we can easily modify the algorithm to also compute $Lpal_w(k)$ online. Since Manacher's algorithm needs to store the length of maximal palindromes for every center in w, it takes $O(n \log n)$ bits of space.

Finally, we show the total number of factors that are removed in the third case of Lemma 3. Once a factor that begins at position j is removed after computing $RLZS(w[1..k])$ for some k, for any $k \leq k' \leq n$, $RLZS(w[1..k'])$ never contains a factor starting at position j. Hence, the total number of factors that are removed in the third case is at most n. This completes the proof. □

3.4.2 Computing $RLZS(w)$ in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space

In this subsection, we present a space efficient algorithm that computes $RLZS(w)$ online, using only $O(n \log \sigma)$ bits of space. Note that we cannot use the method mentioned in the proof of Theorem 2, as it requires $O(n \log n)$ bits of space. Instead, we maintain a compact representation of all suffix palindromes of each prefix $w[1..k]$ of w, as follows. We describe the definition of *Spals* again. For any string w of length $n \geq 1$, let $Spals(w)$ denote the set of the beginning positions of the palindromic suffixes of w, i.e.,

$$Spals(w) = \{n - |s| + 1 \mid s \in Suffix(w), s \text{ is a palindrome}\}.$$

Proposition 2 implies that $Spals(w)$ can be represented by $O(\log^2 n)$ bits of space.

Lemma 4. *We can maintain $O(\log^2 n)$-bit representation of $Spals(w[1..k])$ online for every $1 \leq k \leq n$ in a total of $O(n \log n)$ time.*

Proof. We show how to efficiently update $Spals(w[1..k-1])$ to $Spals(w[1..k])$. Let S be any subset of $Spals(w[1..k-1])$ which is represented by a single arithmetic progression $\langle t, q, m \rangle$, where t is the first (minimum) element, q is the step, and m is the number of elements of the progression. Let s_j be the jth smallest element of S, with $1 \leq j \leq m$. By definition, s_j is a suffix palindrome of $w[1..k-1]$ for any j. In addition, if $m \geq 3$, then it appears that, for any $1 \leq j < m$, s_j has a period q. Therefore, we can test whether the elements of S correspond to the suffix palindromes of $w[1..k]$, by two character comparisons: $w[t-1] = w[k]$

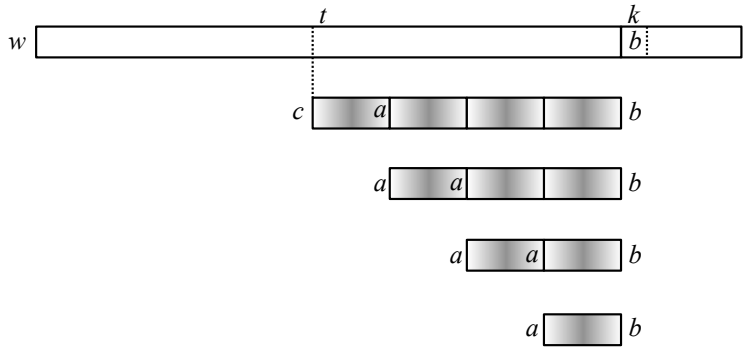

Figure 3.5: Illustration of Lemma 4. Let $w[t-1] = c$, $w[t+q-1] = a$, and $w[k] = b$. $w[t-1..k]$ is a suffix palindrome of $w[1..k]$ iff $c = b$, and $w[t + iq - 1..k]$ is a suffix palindrome of $w[1..k]$ for any $1 \leq i < m$ iff $a = b$.

iff $t - 1 \in \mathit{Spals}(w[1..k])$, and $w[t + q - 1] = w[k]$ iff $t + iq - 1 \notin \mathit{Spals}(w[1..k])$ for any $1 \leq i < m$. (See also Figure 3.5.) If the extension of only one element of S becomes an element of $\mathit{Spals}(w[1..k])$, then we check if it can be merged to the adjacent arithmetic progression that contains closest smaller positions. As above, we can process each arithmetic progression in $O(1)$ time. By Proposition 2, there are $O(\log n)$ arithmetic progressions in $\mathit{Spals}(w[1..k])$ for each prefix of $w[1..k]$ of w. Consequently, for each $1 \leq k \leq n$ we can maintain $O(\log^2 n)$-bit representation of $\mathit{Spals}(w[1..k])$ in a total of $O(n \log n)$ time. □

The main result of this subsection follows:

Theorem 3. *Given a string w of length n, we can compute $RLZS(w)$ in an online manner, in $O(n \log^2 n)$ time and $O(n \log \sigma)$ bits of space.*

Proof. Assume that we are computing a new factor that begins at position l of w. First, we use the algorithm of Theorem 1 and obtain the longest prefix f of $w[l..n]$ such that f^{rev} has an occurrence in $w[1..l-1]$. Then we apply Lemma 3 for $w[1..l + |f| - 1]$, and if the third case holds, then we compute the self-reference factor. We use Lemma 4 to compute $\mathit{Lpal}_w(k)$ for any given position k. After computing the new factor, then we update the suffix tree of the meta-string, and proceed to computing the next factor. Overall, the algorithm takes $O(n \log^2 n)$ time and $O(n \log \sigma + \log^2 n) = O(n \log \sigma)$ bits of space. □

3.5 Reversed LZ factorization and smallest grammar

For any string w, the number of the LZ77 factors [78] (with/without self-references) of w is known to be a lower bound of the smallest grammar that derives only w [67]. Here we briefly show that this is not the case with the reversed LZ factorization (for either with or without self-references).

Theorem 4. *For $\sigma = 3$, there is an infinite series of strings for which the smallest grammar has size $O(\log n)$ while the size of the reversed LZ factorization is $O(n)$.*

Proof. Let $w = (abc)^{\frac{n}{3}}$. Then, $RLZ(w) = RLZS(w) = a, b, c, a, b, c, \ldots, a, b, c$, consisting of exactly n factors. On the other hand, it is easy to see that there exists a grammar of size $O(\log n)$ that generates only w. This completes the proof. □

The above theorem applies to any constant alphabet of size at least 3. When $\sigma = 1$, the size of the smallest grammar and the number of factors in $RLZ(w)$ are both $O(\log n)$, while the number of factors in $RLZS(w)$ is $O(1)$. The binary case where $\sigma = 2$ is open.

Chapter 4

Palindromic Factorization

In this chapter, we introduce a variant of the smallest-sized maximal palindromic factorization problem, named the smallest-sized palindromic factorization problem. We present an efficient online algorithm that computes a smallest-sized palindromic factorization of a given string, and then show how to extend it to the problem of computing a smallest-sized maximal palindromic factorization. We also present an online algorithm that computes a smallest-sized palindromic cover of a given string.

The results in this chapter were originally published in [43].

4.1 Palindromic factorization and palindromic cover of string

A factorization $f_1, \ldots, f_k$ of a string w is said to be a *palindromic factorization* of w if every factor f_i is a palindrome. A palindromic factorization $f_1, \ldots, f_k$ of w is said to be a *maximal palindromic factorization* of w if each f_i is the maximal palindrome at center $|f_1 \cdots f_{i-1}| + \frac{|f_i|+1}{2}$ in w. Since any single character $a \in \Sigma$ is a palindrome, any string has a palindromic factorization. On the other hand, there is a sequence of strings that have no maximal palindromic factorizations, e.g., string $\mathtt{a}(\mathtt{baca})^k$ with $k \geq 1$ has no maximal palindromic factorization.

For a positive integer n, let $[1, n] = \{1, \ldots, n\}$. A set $\{[b_1, e_1], \ldots, [b_h, e_h]\}$ of subintervals of $[1, n]$ is called a *cover* of interval $[1, n]$, if $\bigcup_{i=1}^{h} [b_i, e_i] = [1, n]$. The *size* of the cover is the number h of subintervals in it. A cover $\{[b_1, e_1], \ldots, [b_h, e_h]\}$ of $[1, n]$ is said to be a *palindromic cover* of string w of length n, if $w[b_i..e_i]$ is a palindrome for every i with $1 \leq i \leq h$. Note that any palindromic factorization of string w is a palindromic cover of w, and hence any string has a palindromic cover.

In this chapter, we give efficient solutions to the following problems.

Problem 1. *Given a string w of length n, compute a smallest-sized palindromic factorization of $w[1..i]$ for every $i = 1, \ldots, n$.*

Problem 2. *Given a string w of length n, compute a smallest-sized maximal palindromic factorization of $w[1..i]$ for every $i = 1, \ldots, n$.*

Problem 3. *Given a string w of length n, compute a smallest-sized maximal palindromic cover of $w[1..i]$ for every $i = 1, \ldots, n$.*

To solve the above problems efficiently, we make use of the following known results on palindromes and advanced data structures.

Lemma 5 ([60])**.** *Given a string w of length n, we can compute the maximal palindromes for all centers in w in $O(n)$ time.*

For any two nodes u and v in the same path of a weighted rooted tree, let $\min(u, v)$ be a query that returns a node in the path with minimum weight.

Lemma 6 ([4])**.** *Under a word RAM model, a dynamic tree can be maintained in linear space so that a* $\min$ *query and an operation of adding a leaf to the tree are both supported in the worst-case $O(1)$ time.*

4.2 Combinatorial properties of palindromic suffixes

Here we introduce some combinatorial properties of palindromic suffixes as well as some notations which will be used for designing online algorithms in Sections 4.3 and 4.4. Some of the properties were stated in [6, 62] in a different form.

The next lemma shows a mutual relation between palindromic suffixes and periods.

Lemma 7. *For any palindrome x, $z \in Spals(x)$ iff $|x| - |z|$ is a period of x.*

Proof. $\Rightarrow$: Since x is a palindrome and z is a suffix of x, z^{rev} is a prefix of x. Since z is a palindrome, $z^{\mathrm{rev}} = z$ is a prefix and also a suffix of x, which means $|x| - |z|$ is a period of x.

$\Leftarrow$: Since $|x| - |z|$ is a period of x, z is a prefix of x. It follows from $x = x^{\mathrm{rev}}$ that z (prefix of length $|z|$ of x) and z^{rev} (prefix of length $|z|$ of x^{rev}) are equivalent, which means $z \in Spals(x)$. □

Lemma 7 leads to the following lemmas.

Lemma 8. *For any palindrome x, let z be the largest palindromic suffix of x with $|z| < |x|$. Then $|x| - |z|$ is the smallest period of x.*

Lemma 9. *For any string w and $y, z \in Spals(w)$ with $2|z| > |y| > |z|$, the suffix of length $2|z| - |y|$ is also in $Spals(w)$.*

For any string w of length n, let $LSpals(w)$ denote the lengths of the palindromic suffixes of w, i.e., $LSpals(w) = \{|z| \mid z \in Spals(w)\}$. Thanks to Proposition 2, $LSpals(w)$ can be represented by $O(\log n)$ arithmetic progressions. Let $LSpals(w) = \{q_1, q_2, \ldots, q_{|LSpals(w)|}\}$ with $q_1 < q_2 < \cdots < q_{|LSpals(w)|}$. As a consequence of Lemma 9, it is known that the differences between two adjacent elements in $LSpals(w)$ are monotonically non-decreasing.

We consider partitioning $LSpals(w)$ into $O(\log n)$ groups as follows: For any $1 \leq j \leq |LSpals(w)|$, q_j is contained in an *AP-group* of w with common difference d iff $q_j - q_{j-1} = q_{j+1} - q_j$ or $q_{j+1} - q_j = q_{j+2} - q_{j+1}$, where either of the equalities is ignored when $q_{j'}$ is used for $j' < 1$ or $j' > |LSpals(w)|$, respectively. Namely, if there exist more than two consecutive elements with common difference d of $LSpals(w)$, they, except the largest one, belong to the same group. An element of $LSpals(w)$ that does not belong to any AP-group makes a *single-group* that consists only of itself. Each group can be represented by $\langle q, d, q' \rangle$, where q (resp. q') is the smallest (resp. largest) element of the group and d is its common difference, i.e., $\langle q, d, q' \rangle = \{q, q + d, \ldots, q'\}$, where $d = 0$ for single-groups. Let $X(w)$ denote the set of groups of w, that is $LSpals(w) = \bigcup_{\langle q,d,q' \rangle \in X(w)} \{q, q + d, \ldots, q'\}$.

Let $P(w) = \{|w| - q \mid q \in LSpals(w)\}$, which represents the set of positions p s.t. $p + 1$ is the beginning position of a palindromic suffix of w. For any $\langle q, d, q' \rangle \in X$, let $P(w, \langle q, d, q' \rangle)$ denote a subset of P that restricts palindromic suffixes to ones corresponding to $\langle q, d, q' \rangle$, i.e., $P(w, \langle q, d, q' \rangle) = \{|w| - q, |w| - (q + d), \ldots, |w| - q'\}$. The next lemma shows that the characters attached to the left of palindromic suffixes corresponding to $\langle q, d, q' \rangle$ are identical.

Lemma 10. *For any string w, $\langle q, d, q' \rangle \in X(w)$ and $p, p' \in P(w, \langle q, d, q' \rangle)$, $w[p] = w[p']$.*

Proof. Since it is clear when $|\langle q, d, q' \rangle| = 1$, consider the case where $|\langle q, d, q' \rangle| > 1$. It follows from $\langle q, d, q' \rangle \in X(w)$ and the definition of $X(w)$, $q' + d, q' \in LSpals(w)$. By Lemma 7, d is a period of the suffix z of length $q' + d$ of w. Since positions p, p' are in z and $|p - p'|$ is dividable by d, the lemma holds. □

Lemma 11. *Let w be a string of length n and $a \in \Sigma$. Given a sorted list of $X(w)$, we can compute a sorted list of $X(wa)$ in $O(\log n)$ time.*

Proof. A simple but important observation is that for any $w[i..n+1] \in \mathit{Spals}(wa)$ with $i < n$, $w[i+1..n] \in \mathit{Spals}(w)$. Then except for $w[n+1]$ and $w[n..n+1]$ we can compute $\mathit{Spals}(wa)$ by expanding palindromic suffixes of w.

After adding $w[n+1]$ and $w[n..n+1]$ (if $w[n] = a$) to a tentative list X of $X(wa)$, we process $\langle q, d, q' \rangle \in X(w)$ in increasing order of their lengths. Thanks to Lemma 10, we can process each $\langle q, d, q' \rangle$ in $O(1)$ time, that is, we add $\langle q+2, d, q'+2 \rangle$ to X iff $w[n-q] = a$. After processing all groups in $X(w)$ we check the consecutive groups in X and merge them into one AP-group if needed. Therefore we can get a sorted list of $X(wa)$ in $O(|X(w)|) = O(\log n)$ time. □

4.3 Computing smallest-sized palindromic factorizations online

In this section, we present an $O(n \log n)$-time online algorithm that solves Problem 1 of computing a smallest-sized palindromic factorization of every prefix of a string w of length n. More precisely, we compute an array F such that for each position $1 \leq i \leq n$, $i - F[i]$ gives the length of the last factor of a smallest-sized palindromic factorization of $w[1..i]$ (when more than one factorization exists, choose arbitrary one). Notice that using F, given any position $1 \leq i \leq n$ one can compute a smallest-sized palindromic factorization of $w[1..i]$ by computing the lengths of the factors from right to left in $O(k_i)$ time, where k_i is the number of factors. Our algorithm will also compute k_i's online.

We use the following abbreviation. For any $1 \leq i \leq n$, let $L_i = \mathit{LSpals}(w[1..i])$, $X_i = X(w[1..i])$ and $P_i = P(w[1..i])$. Also, for any $\langle q, d, q' \rangle \in X_i$ let $P_i\langle q, d, q' \rangle = P(w[1..i], \langle q, d, q' \rangle)$.

Suppose that we have processed positions $1, \ldots, i-1$ and now processing i. Note that $F[i] = \arg\min_{p \in P_i}\{k_p\}$ and $k_i = \min_{p \in P_i}\{k_p + 1\}$, where $k_0 = 0$ for convenience. However checking all elements in P_i to compute the minimum value will take $O(i)$ time since $|P_i| = O(i)$ in the worst case.

In order to achieve our aim, we utilize X_i representation. Now we focus on the following subproblem: given $\langle q, d, q' \rangle \in X_i$, compute $\arg\min_{p \in P_i\langle q,d,q' \rangle}\{k_p\}$. We show how to solve this in constant time. When $|\langle q, d, q' \rangle|$ is small enough as we can treat it as a constant, we can solves this in constant time naïvely. However $|\langle q, d, q' \rangle| = O(i)$ in the worst case. In what follows, we

consider how to process efficiently the case where $|\langle q, d, q' \rangle|$ is large. The next lemma gives a key observation.

Lemma 12. *For any $\langle q, d, q' \rangle \in X_i$ with $|\langle q, d, q' \rangle| \geq 3$, $\langle q, d, q' - d \rangle \in X_{i-d}$.*

Proof. First we show $q, q + d, \ldots, q'$ is a subsequence of L_{i-d}. By definition of X_i, the suffix z of length $q' + d$ of $w[1..i]$ is a palindrome. It follows from Lemma 7 that d is a period of z, which means that palindromic structures of $w[i - |z| + d + 1..i]$ are identical to those of $w[i - |z| + 1..i - d]$. In other words, $L_i \cap [1, q'] = L_{i-d} \cap [1, q']$. Hence $q, q + d, \ldots, q'$ is a subsequence of L_{i-d}. Another consequence of the equality $L_i \cap [1, q'] = L_{i-d} \cap [1, q']$ is that the largest element of L_{i-d} which is smaller than q is not $q - d$.

Next we show that $q' + d \notin L_{i-d}$. Assume on the contrary that $q' + d \in L_{i-d}$, i.e., the suffix y of length $q' + d$ of $w[1..i - d]$ is a palindrome. It follows from Lemma 7 that d is a period of y. Let x be the suffix of length q' of y. Note that x is also a prefix of z. Let $y = y'x$ and $z = xz'$. Since x is a palindrome and has y' as a prefix and z' as a suffix, $y' = (z')^{\mathrm{rev}}$. Therefore $y'xz'$, which is the suffix of length $q' + 2d$ of $w[1..i]$, is a palindrome. This implies that $q, q + d, \ldots, q' + d, q' + 2d$ is a subsequence of L_i, which contradicts that $\langle q, d, q' \rangle \in X_i$.

Putting all together, $q, q + d, \ldots, q'$ with $|\{q, q + d, \ldots, q'\}| \geq 3$ is a subsequence of L_{i-d} that is maximal with common difference d, which leads to the argument. □

Since $P_{i-d}\langle q, d, q' - d \rangle = \{i - d - q, i - 2d - q, \ldots, i - q'\} = P_i\langle q, d, q' \rangle \setminus \{i - q\}$, Lemma 12 implies that at position $i - d$ we actually computed $p' = \arg\min_{p \in P_i\langle q,d,q' \rangle \setminus \{i-q\}} \{k_p\}$. Then if we keep this information, it suffices for us to compare $k_{p'}$ and k_{i-q}, and take the smaller one, which requires constant time.

More precisely, we maintain the following data structures dynamically: If there exists $\langle q, d, q' \rangle \in X_i$ with $|\langle q, d, q' \rangle| \geq 5$, our algorithm maintains an array A_d of length d s.t. for $j \equiv i \pmod d$,

just before processing position i: $A_d[j]$ is undefined if $|\langle q, d, q' \rangle| = 5$, and $A_d[j] = \arg\min_{p \in P_{i-d}\langle q,d,q'-d \rangle} \{k_p\}$ if $|\langle q, d, q' \rangle| \geq 6$.

just after processing position i: $A_d[j]$ is updated to be $\arg\min_{p \in P_i\langle q,d,q' \rangle} \{k_p\}$.

This array A_d is used for AP-groups $\langle q, d, q' \rangle \in X$ with $|\langle q, d, q' \rangle| \geq 5$ of adjacent positions. It is allocated when necessary and discarded as soon as an AP-group with common difference d disappears from X while processing positions.

The next lemma ensures that when A_d is allocated there exists AP-groups with common difference d for the d immediately preceding positions of i, which will be used for analyzing the cost for allocation/deallocation of A_d arrays.

Lemma 13. *Let $\langle q, d, q' \rangle \in X_i$ with $|\langle q, d, q' \rangle| \geq 5$. For any $1 \leq h \leq d$, there exists an AP-group of $w[1..i-h]$ with common difference d.*

Proof. We show that there exist at least three consecutive palindromic suffixes of $w[1..i-h]$ with common difference d. From the assumption, there exist at least six palindromic suffixes of $w[1..i]$ with common difference d. Let us take a look at the largest three of them of lengths $q'+d, q'$ and $q'-d$, i.e., $w[i-(q'+d)+1..i]$, $w[i-q'+1..i]$ and $w[i-(q'-d)+1..i]$. Since $q'-d > 3d$, we can obtain three palindromes that ends at $i-h$, $x = w[i-(q'+d)+1+h..i-h]$, $y = w[i-q'+1+h..i-h]$ and $z = w[i-(q'-d)+1+h..i-h]$. Notice that $|x|-|y| = |y|-|z| = d$ and $|z| > d$.

Now we will show that there is no palindromic suffix u of $w[1..i-h]$ s.t. $|x| > |u| > |y|$ or $|y| > |u| > |z|$. Assume on the contrary that such u exists. Let $d' = |u| - |y|$ if $|u| > |y|$ and $d' = |u| - |z|$ if $|y| > |u|$. Let v be the prefix of length d of z. It follows from Lemma 7 that d' and $d - d'$ are periods of v. From the periodicity lemma (see Proposition 1), $\gcd(d', d-d')$ is also a period of v. However, Lemma 8 implies that the suffix of length $q'+d$ of $w[1..i]$ has the smallest period d, and hence its substring of length d cannot have a period $d''(< d)$ which can divide d, a contradiction. □

Theorem 5. *Problem 1 can be solved in $O(n \log n)$ time and $O(n)$ space in an online manner.*

Proof. Suppose that we have processed positions $1, \ldots, i-1$, i.e., we now have X_{i-1} and A_d arrays are properly maintained. At the beginning of processing i, we compute X_i from X_{i-1} as well as allocate/deallocate A_d arrays if necessary. Next, for each $\langle q, d, q' \rangle \in X_i$ we compute $\arg\min_{p \in P_i \langle q,d,q' \rangle} \{k_p\}$ by naïvely checking if $|\langle q, d, q' \rangle| \leq 5$, and using A_d arrays if $|\langle q, d, q' \rangle| \geq 6$. If $|\langle q, d, q' \rangle| \geq 5$, we also update its corresponding value of A_d. Then $F[i]$ and k_i can be obtained from $\arg_p \min_{\langle q,d,q' \rangle \in X_i, p \in P_i \langle q,d,q' \rangle} \{k_p\}$.

By Lemma 11, computing X_i for all positions $1 \leq i \leq n$ takes a total of $O(n \log n)$ time. For each $\langle q, d, q' \rangle \in X_i$, $\arg\min_{p \in P_i \langle q,d,q' \rangle} \{k_p\}$ can be computed in constant time by using A_d arrays, and hence it takes $O(n \log n)$ time in total. It follows from Lemma 13 that the cost for allocation/deallocation of each position of A_d arrays can be attributed to distinct $\langle q, d, q' \rangle \in X_i$ for some position i. Therefore the algorithm runs in $O(n \log n)$ time.

During the computation we maintain A_d arrays dynamically so that for any position i its space usage is bounded by $\sum\{d \mid \langle q, d, q' \rangle \in X_i\} = O(i) = O(n)$. □

4.4 Computing smallest-sized maximal palindromic factorizations online

In this section, we present an $O(n \log n)$-time online algorithm that solves Problem 2 of computing a smallest-sized maximal palindromic factorization of every prefix of a string w of length n. Our high-level strategy is similar to that of previous section, i.e., checking palindromic suffixes of $w[1..i]$ for each position $1 \leq i \leq n$. However, in this problem we have to be careful not to use a non-maximal palindrome as a factor.

For any position $1 \leq i \leq n$ let h_i be the number of factors of a maximal palindromic factorization of $w[1..i]$ of the smallest size if such exists, and ∞ otherwise. For any position $1 \leq i < n$, let h'_i be the number of factors in a smallest-sized palindromic factorization of $w[1..i]$ which consists only of maximal palindromes of $w[1..i+1]$ if such exists, and ∞ otherwise. Note that the values of h_i and h'_i may differ, since we can use any palindromic suffix of $w[1..i]$ for h_i while for h'_i we cannot use it if it is not maximal in $w[1..i+1]$.

For any position $1 \leq i < n$ let P'_i denote the set of positions p s.t. $w[p+1..i]$ is a maximal palindrome of w, that is $P'_i = \{p \in P_i \mid w[p] \neq w[i+1]\}$. It holds that $h_i = \min_{p \in P_i}\{h'_p + 1\}$ and $h'_i = \min_{p \in P'_i}\{h'_p + 1\}$, where $\infty + 1 = \infty$. Similar to array F of Section 4.3, in this problem we compute two arrays G and G' s.t. for each position i, $G[i] = \arg\min_{p \in P_i}\{h'_p\}$ and $G'[i] = \arg\min_{p \in P'_i}\{h'_p\}$, where $h'_0 = 0$ for convenience. Using these arrays, given any position $1 \leq i \leq n$, one can compute a maximal palindromic factorization of $w[1..i]$ of the smallest size (if $h_i \neq \infty$) in $O(h_i)$ time as follows; the h_ith factor is $w[G[i]+1..i]$, and for any $1 < j \leq h_i$ the $(j-1)$th factor is $w[G'[i_j]+1..i_j]$, where $i_j + 1$ is the beginning position of the jth factor.

In light of Lemma 10, for any $\langle q, d, q' \rangle \in X_i$ we can know in constant time if this group corresponds to P'_i or not by just investigating $w[p]$ and $w[i+1]$ for arbitrary position $p \in P_i\langle q, d, q' \rangle$. Hence during the computation of arrays G and G' we can process each $\langle q, d, q' \rangle \in X_i$ in constant time.

Everything else can be managed in the same way as the algorithm proposed in Section 4.3. As to the maintenance of A_d arrays, a minor remark is that we use h'_p instead of k_p, i.e., the

value of A_d is updated to be $\arg\min_{p \in P_i\langle q,d,q'\rangle}\{h'_p\}$. Therefore we get the following theorem.

Theorem 6. *Problem 2 can be solved in $O(n \log n)$ time and $O(n)$ space in an online manner.*

4.5 Computing smallest-sized palindromic covers online

In this section, we present an $O(n)$-time online algorithm that solves Problem 3 of computing a smallest-sized palindromic cover of every prefix of a string w of length n. We begin this section with a simpler problem of computing a smallest-sized palindromic cover of w, and present an $O(n)$-time offline algorithm that solves the problem. The following observation is a key to our solution.

Observation 1. *If $\{[b_1, e_1], \ldots, [b_k, e_k]\}$ is a palindromic cover of string w, then there is a palindromic cover $\{[b'_1, e'_1], \ldots, [b'_k, e'_k]\}$ of w such that $b'_i = b_i - d$ and $e'_i = e_i + d$ for some $d \geq 0$ and $w[b'_i..e'_i]$ is the maximal palindrome at center $\frac{b'_i+e'_i}{2} = \frac{b_i+e_i}{2}$ for all $1 \leq i \leq k$.*

The above observation says that for any palindromic cover of string w, there always exists a palindromic cover of w of the same size which consist only of maximal palindromes. Hence, to compute a palindromic cover of w of the smallest size, it suffices for us to consider covers which are composed only of maximal palindromes.

Theorem 7. *Given a string w of length n, we can compute a smallest-sized palindromic cover of w in $O(n)$ time.*

Proof. We use an array R of length n such that $R[i]$ stores the beginning position of the maximal palindrome that contains position i with the least (leftmost) beginning position. We compute a palindromic cover of w from array R in a greedy manner, from right to left; The longest palindromic suffix of w is the rightmost palindrome of a smallest-sized palindromic cover of w. Given a smallest-sized palindromic cover of suffix $w[b..n]$, we add into the cover the maximal palindrome stored in $R[b-1]$, the maximal palindrome that contains position $b-1$ with the leftmost beginning position. The procedure terminates when we obtain from R a palindromic prefix of w, i.e., a maximal palindrome that begins at position 1 in w.

Let C be the maximal palindromic cover the above algorithm computes. Let C_k denote the subset of C that contains the last k intervals (the k consecutive intervals from right) in C. Let b_k be the beginning position of the leftmost interval of C_k. We show that C is a smallest-sized maximal palindromic cover of w by induction on k. If $k = 1$, then C_1 contains the longest

palindromic suffix of w. Then, clearly $C_1 = \{[b_1, n]\}$ is the smallest-sized palindromic cover of $w[b_1..n]$. Assume that C_k is the smallest-sized palindromic cover of $w[b_k..n]$ for $k \geq 1$, computed greedily as above. Then, $C_{k+1} = [R[b_k - 1], e] \cup C_k$, where e denotes the ending position of the corresponding maximal palindrome starting at $R[b_k - 1]$. By definition of $R[b_k - 1]$, C_{k+1} is a smallest-sized palindromic cover of $w[b_{k+1}..n]$. Since $b_{|C|} = 1$, $C_{|C|} = C$ is a smallest-sized palindromic cover of $w[b_{|C|}..n] = w$.

Let us analyze the time complexity of the algorithm. The maximal palindromes can be computed in $O(n)$ time by Lemma 5. To compute array R, we sort the maximal palindromes in increasing order of their beginning positions, and consider only the longest one for each beginning position. This can be done in $O(n)$ time by using bucket sort. Then R can be obtained in $O(n)$ time from the maximal palindromes above. Obviously it takes $O(n)$ time to greedily choose the maximal palindromes from R. This completes the proof. □

Now, we extend the algorithm of Theorem 7 to compute a smallest palindromic cover of every prefix of a given string in an online manner (Problem 5). A basic idea is to compute the longest palindromic suffix of each prefix $w[1..i]$ of a given string w, which is the last (rightmost) palindrome of a smallest-sized palindromic cover of $w[1..i]$.

Theorem 8. *Given a string w of length n, we can compute an $O(n)$-size representation of smallest-sized palindromic covers of all prefixes of w in $O(n)$ time. Given a position j ($1 \leq j \leq n$) in w, the representation returns the size s_j of a smallest-sized palindromic cover of $w[1..j]$ in $O(1)$ time, and allows us to compute a smallest-sized palindromic cover in $O(s_j)$ time.*

Proof. For any $1 \leq i \leq n$, let l_i denote the length of the longest palindromic suffix of $w[1..i]$, and let s_i be the size of a smallest-sized palindromic cover of $w[1..i]$. If $i = 1$, then clearly $[1, 1]$ is the smallest-sized palindromic cover of $w[1..1] = w[1]$. Hence $s_1 = 1$ and $l_1 = 1$. Consider the case where $i \geq 2$. There are two cases to consider.

- When $|l_i| = |l_{i-1}| + 2$, namely, the longest palindromic suffix of $w[1..i]$ is an extension of that of $w[1..i]$. In this case, $s_i = \min\{s_{i-1}, s_{i-l_i} + 1\}$. (See also Figure 4.1)
- When $|l_i| < |l_{i-1}|+2$, namely, the longest palindromic suffix of $w[1..i]$ is not an extension of that of $w[1..i]$. In this case, $s_i = \min\{s_k \mid i - l_i \leq k < i\} + 1$.

For all $1 \leq i \leq n$, we can compute l_i in $O(n)$ time in an online manner in increasing order of i [71]. We also compute s_i in increasing order of i. When $|l_i| = |l_{i-1}| + 2$, we can easily

Figure 4.1: To the left is the case where $s_i = s_{i-1}$, and to the right is the case where $s_i = s_{i-l_i} + 1$.

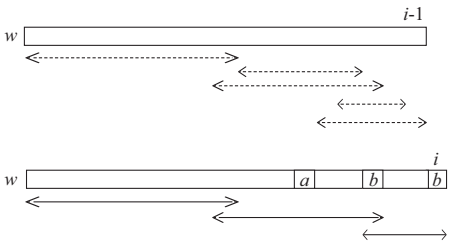

Figure 4.2: The case where $s_i = \min\{s_k \mid i - l_i \leq k < i\} + 1$.

compute s_i in $O(1)$ time. To efficiently compute s_i also in the case where $|l_i| < |l_{i-1}| + 2$, we maintain a list such that its ith element is s_i. Now, given l_i and the list that stores s_k for all $1 \leq k < i$, which is augmented with the data structure of Lemma 6, each s_i can be computed in $O(1)$ time. We append s_i to the list and the data structure of Lemma 6 in $O(1)$ time. Hence, the total time complexity is $O(n)$.

To answer a smallest-sized palindromic cover of a prefix $w[1..j]$ for a given position $1 \leq j \leq n$ in $O(s_j)$ time, we do the following. For each $1 \leq i \leq n$ let $b_i = i - l_i + 1$, i.e., b_i is the beginning position of the longest palindromic suffix of $w[1..i]$. We maintain a list that represents b_i for all $1 \leq i \leq n$ in an online manner, which is also augmented with the data structure of Lemma 6. After computing l_i and b_i for position i, we compute $r = \min\{b_k \mid b_i - 1 \leq k < i\}$ in $O(1)$ time by Lemma 6. Then, we have $r = R[b_i]$, where R is the array used in Theorem 7. Therefore, given a position j, we can compute a smallest-sized palindromic cover of $w[1..j]$ from right to left, in $O(s_j)$ time. □

4.6 Related Work

We mention two important related works. In 2014 Fici et al. [32] develeped an $O(n \log n)$-time algorithm for the smallest sized palindromic factoriazation problem, independently of our work. The algorithm is essentially the same as ours. In 2017 Borozdin et al. [13] presented the first

linear time solution to the problem. Their algorithm is based on our $O(n \log n)$-time algorithm and utilizes the so-called four Russian technique.

Chapter 5

Diverse Palindromic Factorization is NP Complete

A diverse palindromic factorization of a string w is a palindromic factorization of w such that the factors are mutually distinct. In this chapter, we prove that the existence problem of the diverse palindromic factorization is NP-complete, by showing a reduction from the circuit satisfiability problem [58].

The results in this chapter were originally published in [11].

5.1 Outline of the proof

In complexity theory, a Boolean circuit is formally a directed acyclic graph in which each node is either a source or one of a specified set of logic gates. The gates are usually AND, OR and NOT, with AND and OR gates each having in-degree at least 2 and NOT gates each having in-degree 1. A gate's predecessors and successors are called its inputs and outputs, and sources and sinks are called the circuit's inputs and outputs. A circuit with a single output is said to be satisfiable if and only if it is possible to assign each gate a value true or false such that the output is true and all the gates' semantics are respected: e.g., each AND gate is true if and only if all its inputs are true, each OR gate is true if and only if at least one of its inputs is true, and each NOT gate is true if and only if its unique input is false. Notice that with these semantics, a truth assignment to the circuit's inputs determines the truth values of all the gates.

The circuit satisfiability problem [58] (see also, e.g., [35]) is to determine whether a given single-output Boolean circuit C is satisfiable. It was one of the first problems proven NP-complete and is often the first such problem taught in undergraduate courses. We will show

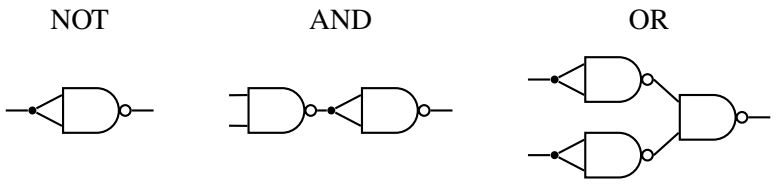

Figure 5.1: Construction of NOT, AND and OR gates using NAND gates.

how to build, in time linear in the size of C, a string that has a diverse palindromic factorization if and only if C is satisfiable. It follows that diverse palindromic factorization is also NP-hard. Our construction is similar to the Tseitin Transform [73] from Boolean circuits to CNF formulas.

We can make each AND or OR gate's in-degree 2 and each gate's out-degree 1 at the cost of at most a logarithmic increase in the size and depth of the circuit, using splitter gates with one input and two outputs that should have the same truth value as the input. A NAND gate is true if and only if at least one of its inputs is false. AND, OR and NOT gates can be implemented with a constant number of NAND gates (see Figure 5.1), so we assume without loss of generality that C is composed only of NAND gates with two inputs and one output each and splitter gates. Boolean circuits are a model for real circuits, so henceforth we assume the gates' semantics are respected, call the graph's edges wires, say each splitter divides one wire in two, and discuss wires' truth values instead of discussing the truth values of the gates at which those wires originate.

We assume each wire in C is labelled with a unique symbol (considering a split to be the end of an incoming wire and the beginning of two new wires, so all three wires have different labels). For each such symbol a, and some auxiliary symbols we introduce during our construction, we use as characters in our construction three related symbols: a itself, $\bar{a}$ and x_a. We indicate an auxiliary symbol related to a by writing a' or a''. We write x_a^j to denote j copies of x_a. We emphasize that, despite their visual similarity, a and $\bar{a}$ are separate characters, which play complementary roles in our reduction. We use $\$$ and $\#$ as generic separator symbols, which we consider to be distinct for each use; to prevent confusion, we add different superscripts to their different uses within the same part of the construction.

We can build a sequence $C_0, \ldots, C_t$ of subcircuits such that C_0 is empty, $C_t = C$ and, for $1 \leq i \leq t$, we obtain C_i from C_{i-1} by one of the following operations (see Figure 5.2 for an example):

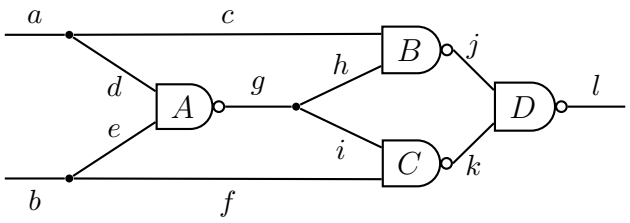

Figure 5.2: To construct the circuit above (computing XOR) we need to add wires a and b, split a into c and d, split b into e and f, add gate A, split g into h and i, and finally add gates B, C and D.

- adding a new wire (which is both an input and an output in C_i),
- splitting an output of C_{i-1} into two outputs,
- making two outputs of C_{i-1} the inputs of a new NAND gate.

We will show how to build in time linear in the size of C, inductively and in turn, a sequence of strings $S_1, \ldots, S_t$ such that S_i represents C_i according to the following definitions:

Definition 5. *A diverse palindromic factorization P of a string S_i* encodes *an assignment τ to the inputs of a circuit C_i if the following conditions hold:*

- *if τ makes an output of C_i labelled a true, then a, x_a and $x_a\bar{a}x_a$ are complete factors in P but $\bar{a}$, x_aax_a and x_a^j are not for $j > 1$;*
- *if τ makes an output of C_i labelled a false, then $\bar{a}$, x_a and x_aax_a are complete factors in P but a, $x_a\bar{a}x_a$ and x_a^j are not for $j > 1$;*
- *if a is a label in C but not in C_i, then none of a, $\bar{a}$, x_aax_a, $x_a\bar{a}x_a$ and x_a^j for $j \geq 1$ are complete factors in P.*

Definition 6. *A string S_i* represents *a circuit C_i if each assignment to the inputs of C_i is encoded by some diverse palindromic factorization of S_i, and each diverse palindromic factorization of S_i encodes some assignment to the inputs of C_i.*

Once we have S_t, we can easily build in constant time a string S that has a diverse palindromic factorization if and only if C is satisfiable. To do this, we append $\$\#\, x_aax_a$ to S_t, where $\$$ and $\#$ are symbols not occurring in S_t and a is the label on C's output. Since $\$$ and $\#$ do not occur in S_t and occur as a pair of consecutive characters in S, they must each be complete

factors in any palindromic factorization of S. It follows that there is a diverse palindromic factorization of S if and only if there is a diverse palindromic factorization of S_t in which $x_a a x_a$ is not a factor, which is the case if and only if there is an assignment to the inputs of C that makes its output true.

5.2 Adding a wire

Suppose C_i is obtained from C_{i-1} by adding a new wire labelled a. If $i = 1$ then we set $S_i = x_a a x_a \bar{a} x_a$, whose two diverse palindromic factorizations $x_a,\ a,\ x_a \bar{a} x_a$ and $x_a a x_a,\ \bar{a},\ x_a$ encode the assignments true and false to the wire labelled a, which is both the input and output in C_i. If $i > 1$ then we set

$$S_i = S_{i-1}\,\$\#\,x_a a x_a \bar{a} x_a\,,$$

where \$ and $\#$ are symbols not occurring in S_{i-1} and not equal to a', $\overline{a'}$ or $x_{a'}$ for any label a' in C.

Since \$ and $\#$ do not occur in S_{i-1} and occur as a pair of consecutive characters in S_i, they must each be complete factors in any palindromic factorization of S_i. Therefore, any diverse palindromic factorization of S_i is the concatenation of a diverse palindromic factorization of S_{i-1} and either $\$,\ \#,\ x_a,\ a,\ x_a \bar{a} x_a$ or $\$,\ \#,\ x_a a x_a,\ \bar{a},\ x_a$. Conversely, any diverse palindromic factorization of S_{i-1} can be extended to a diverse palindromic factorization of S_i by appending either $\$,\ \#,\ x_a,\ a,\ x_a \bar{a} x_a$ or $\$,\ \#,\ x_a a x_a,\ \bar{a},\ x_a$.

Assume S_{i-1} represents C_{i-1}. Let τ be an assignment to the inputs of C_i and let P be a diverse palindromic factorization of S_{i-1} encoding τ restricted to the inputs of C_{i-1}. If τ makes the input (and output) of C_i labelled a true, then P concatenated with $\$,\ \#,\ x_a,\ a,\ x_a \bar{a} x_a$ is a diverse palindromic factorization of S_i that encodes τ. If τ makes that input false, then P concatenated with $\$,\ \#,\ x_a a x_a,\ \bar{a},\ x_a$ is a diverse palindromic factorization of S_i that encodes τ. Therefore, each assignment to the inputs of C_i is encoded by some diverse palindromic factorization of S_i.

Now let P be a diverse palindromic factorization of S_i and let τ be the assignment to the inputs of C_{i-1} that is encoded by a prefix of P. If P ends with $\$,\ \#,\ x_a,\ a,\ x_a \bar{a} x_a$ then P encodes the assignment to the inputs of C_i that makes the input labelled a true and makes the other inputs true or false according to τ. If P ends with $\$,\ \#,\ x_a a x_a,\ \bar{a},\ x_a$ then P encodes the assignment to the inputs of C_i that makes the input labelled a false and makes the other inputs

true or false according to τ. Therefore, each diverse palindromic factorization of S_i encodes some assignment to the inputs of C_i.

Lemma 14. *We can build a string S_1 that represents C_1. If we have a string S_{i-1} that represents C_{i-1} and C_i is obtained from C_{i-1} by adding a new wire, then in constant time we can append symbols to S_{i-1} to obtain a string S_i that represents C_i.*

5.3 Splitting a wire

Now suppose C_i is obtained from C_{i-1} by splitting an output of C_{i-1} labelled a into two outputs labelled b and c. We set

$$S'_i = S_{i-1}\,\$\#\,x_a^3 b' x_a a x_a c' x_a^5\,\$'\#'\,x_a^7 \overline{b'} x_a \bar{a} x_a \overline{c'} x_a^9\,,$$

where \$, \$′, #, #′, b', $\overline{b'}$, c' and $\overline{c'}$ are symbols not occurring in S_{i-1} and not equal to a', $\overline{a'}$ or $x_{a'}$ for any label a' in C.

Since \$, \$′, # and #′ do not occur in S_{i-1} and occur as pairs of consecutive characters in S'_i, they must each be complete factors in any palindromic factorization of S'_i. Therefore, a simple case analysis shows that any diverse palindromic factorization of S'_i is the concatenation of a diverse palindromic factorization of S_{i-1} and one of

$$\begin{aligned}
&\$,\ \#,\ x_a^3,\ b',\ x_a a x_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^2,\ x_a^4,\ x_a \overline{b'} x_a,\ \bar{a},\ x_a \overline{c'} x_a,\ x_a^8\,,\\
&\$,\ \#,\ x_a^3,\ b',\ x_a a x_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^4,\ x_a^2,\ x_a \overline{b'} x_a,\ \bar{a},\ x_a \overline{c'} x_a,\ x_a^8\,,\\
&\$,\ \#,\ x_a^3,\ b',\ x_a a x_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^6,\ x_a \overline{b'} x_a,\ \bar{a},\ x_a \overline{c'} x_a,\ x_a^8\,,\\
&\$,\ \#,\ x_a^2,\ x_a b' x_a,\ a,\ x_a c' x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a \bar{a} x_a,\ \overline{c'},\ x_a^3,\ x_a^6\,,\\
&\$,\ \#,\ x_a^2,\ x_a b' x_a,\ a,\ x_a c' x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a \bar{a} x_a,\ \overline{c'},\ x_a^6,\ x_a^3\,,\\
&\$,\ \#,\ x_a^2,\ x_a b' x_a,\ a,\ x_a c' x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a \bar{a} x_a,\ \overline{c'},\ x_a^9\,;
\end{aligned}$$

In any diverse palindromic factorization of S'_i, therefore, either b' and c' are complete factors but $\overline{b'}$ and $\overline{c'}$ are not, or vice versa.

Conversely, any diverse palindromic factorization of S_{i-1} in which a, x_a and $x_a \bar{a} x_a$ are complete factors but $\bar{a}$, $x_a a x_a$ and x_a^j are not for $j > 1$, can be extended to a diverse palindromic

factorization of S'_i by appending either of

$$\$,\ \#,\ x_a^3,\ b',\ x_aax_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^2,\ x_a^4,\ x_a\overline{b'}x_a,\ \bar{a},\ x_a\overline{c'}x_a,\ x_a^8\,,$$
$$\$,\ \#,\ x_a^3,\ b',\ x_aax_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^6,\ x_a\overline{b'}x_a,\ \bar{a},\ x_a\overline{c'}x_a,\ x_a^8\,;$$

any diverse palindromic factorization of S_{i-1} in which $\bar{a}$, x_a and x_aax_a are complete factors but a, $x_a\bar{a}x_a$ and x_a^j are not for $j > 1$, can be extended to a diverse palindromic factorization of S'_i by appending either of

$$\$,\ \#,\ x_a^2,\ x_ab'x_a,\ a,\ x_ac'x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a\bar{a}x_a,\ \overline{c'},\ x_a^3,\ x_a^6\,,$$
$$\$,\ \#,\ x_a^2,\ x_ab'x_a,\ a,\ x_ac'x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a\bar{a}x_a,\ \overline{c'},\ x_a^9\,.$$

We set

$$S_i = S'_i\,\$''\#''\,x_bbx_bb'x_b\overline{b'}x_b\bar{b}x_b\,\$'''\#'''\,x_ccx_cc'x_c\overline{c'}x_c\bar{c}x_c\,,$$

where $\$''$, $\$'''$, $\#''$ and $\#'''$ are symbols not occurring in S'_i and not equal to a', $\overline{a'}$ or $x_{a'}$ for any label a' in C. Since $\$''$, $\$'''$, $\#''$ and $\#'''$ do not occur in S'_i and occur as pairs of consecutive characters in S'_i, they must each be complete factors in any palindromic factorization of S_i. Therefore, any diverse palindromic factorization of S_i is the concatenation of a diverse palindromic factorization of S'_i and one of

$$\$'',\ \#'',\ x_b,\ b,\ x_bb'x_b,\ \overline{b'},\ x_b\bar{b}x_b,\ \$''',\ \#''',\ x_c,\ c,\ x_cc'x_c,\ \overline{c'},\ x_c\bar{c}x_c\,,$$
$$\$'',\ \#'',\ x_bbx_b,\ b',\ x_b\overline{b'}x_b,\ \bar{b},\ x_b,\ \$''',\ \#''',\ x_ccx_c,\ c',\ x_c\overline{c'}x_c,\ \bar{c},\ x_c\,.$$

Conversely, any diverse palindromic factorization of S'_i in which b' and c' are complete factors but $\overline{b'}$ and $\overline{c'}$ are not, can be extended to a diverse palindromic factorization of S_i by appending

$$\$'',\ \#'',\ x_b,\ b,\ x_bb'x_b,\ \overline{b'},\ x_b\bar{b}x_b,\ \$''',\ \#''',\ x_c,\ c,\ x_cc'x_c,\ \overline{c'},\ x_c\bar{c}x_c\,;$$

any diverse palindromic factorization of S'_i in which $\overline{b'}$ and $\overline{c'}$ are complete factors but b' and c' are not, can be extended to a diverse palindromic factorization of S_i by appending

$$\$'',\ \#'',\ x_bbx_b,\ b',\ x_b\overline{b'}x_b,\ \bar{b},\ x_b,\ \$''',\ \#''',\ x_ccx_c,\ c',\ x_c\overline{c'}x_c,\ \bar{c},\ x_c\,.$$

Assume S_{i-1} represents C_{i-1}. Let τ be an assignment to the inputs of C_{i-1} and let P be a

diverse palindromic factorization of S_{i-1} encoding τ. If τ makes the output of C_{i-1} labelled a true, then P concatenated with, e.g.,

$$\$,\ \#,\ x_a^3,\ b',\ x_aax_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^2,\ x_a^4,\ x_a\overline{b'}x_a,\ \bar{a},\ x_a\overline{c'}x_a,\ x_a^8,$$
$$\$'',\ \#'',\ x_b,\ b,\ x_bb'x_b,\ \overline{b'},\ x_b\bar{b}x_b,\ \$''',\ \#''',\ x_c,\ c,\ x_cc'x_c,\ \overline{c'},\ x_c\bar{c}x_c$$

is a diverse palindromic factorization of S_i. Notice b, c, x_b, x_c, $x_b\bar{b}x_b$ and $x_c\bar{c}x_c$ are complete factors but $\bar{b}$, $\bar{c}$, x_bbx_b, x_ccx_c, x_b^j and x_c^j for $j > 1$ are not. Therefore, this concatenation encodes the assignment to the inputs of C_i that makes them true or false according to τ.

If τ makes the output of C_{i-1} labelled a false, then P concatenated with, e.g.,

$$\$,\ \#,\ x_a^2,\ x_ab'x_a,\ a,\ x_ac'x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a\bar{a}x_a,\ \overline{c'},\ x_a^3,\ x_a^6,$$
$$\$'',\ \#'',\ x_bbx_b,\ b',\ x_b\overline{b'}x_b,\ \bar{b},\ x_b,\ \$''',\ \#''',\ x_ccx_c,\ c',\ x_c\overline{c'}x_c,\ \bar{c},\ x_c$$

is a diverse palindromic factorization of S_i. Notice $\bar{b}$, $\bar{c}$, x_b, x_c, x_bbx_b and x_ccx_c are complete factors but b, c, $x_b\bar{b}x_b$, $x_c\bar{c}x_c$, x_b^j and x_c^j for $j > 1$ are not. Therefore, this concatenation encodes the assignment to the inputs of C_i that makes them true or false according to τ. Since C_{i-1} and C_i have the same inputs, each assignment to the inputs of C_i is encoded by some diverse palindromic factorization of S_i.

Now let P be a diverse palindromic factorization of S_i and let τ be the assignment to the inputs of C_{i-1} that is encoded by a prefix of P. If P ends with any of

$$\$,\ \#,\ x_a^3,\ b',\ x_aax_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^2,\ x_a^4,\ x_a\overline{b'}x_a,\ \bar{a},\ x_a\overline{c'}x_a,\ x_a^8\,,$$
$$\$,\ \#,\ x_a^3,\ b',\ x_aax_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^4,\ x_a^2,\ x_a\overline{b'}x_a,\ \bar{a},\ x_a\overline{c'}x_a,\ x_a^8\,,$$
$$\$,\ \#,\ x_a^3,\ b',\ x_aax_a,\ c',\ x_a^5,\ \$',\ \#',\ x_a^6,\ x_a\overline{b'}x_a,\ \bar{a},\ x_a\overline{c'}x_a,\ x_a^8$$

followed by

$$\$'',\ \#'',\ x_b,\ b,\ x_bb'x_b,\ \overline{b'},\ x_b\bar{b}x_b,\ \$''',\ \#''',\ x_c,\ c,\ x_cc'x_c,\ \overline{c'},\ x_c\bar{c}x_c\,,$$

then a must be a complete factor in the prefix of P encoding τ, so τ must make the output of C_{i-1} labelled a true. Since b, c, x_b, x_c, $x_b\bar{b}x_b$ and $x_c\bar{c}x_c$ are complete factors in P but $\bar{b}$, $\bar{c}$, x_bbx_b, x_ccx_c, x_b^j and x_c^j for $j > 1$ are not, P encodes the assignment to the inputs of C_i that makes them true or false according to τ.

If P ends with any of

$$\$,\ \#,\ x_a^2,\ x_a b' x_a,\ a,\ x_a c' x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a \bar{a} x_a,\ \overline{c'},\ x_a^3,\ x_a^6\,,$$
$$\$,\ \#,\ x_a^2,\ x_a b' x_a,\ a,\ x_a c' x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a \bar{a} x_a,\ \overline{c'},\ x_a^6,\ x_a^3\,,$$
$$\$,\ \#,\ x_a^2,\ x_a b' x_a,\ a,\ x_a c' x_a,\ x_a^4,\ \$',\ \#',\ x_a^7,\ \overline{b'},\ x_a \bar{a} x_a,\ \overline{c'},\ x_a^9$$

followed by

$$\$'',\ \#'',\ x_b b x_b,\ b',\ x_b \overline{b'} x_b,\ \bar{b},\ x_b,\ \$''',\ \#''',\ x_c c x_c,\ c',\ x_c \overline{c'} x_c,\ \bar{c},\ x_c\,,$$

then $\bar{a}$ must be a complete factor in the prefix of P encoding τ, so τ must make the output of C_{i-1} labelled a false. Since $\bar{b}$, $\bar{c}$, x_b, x_c, $x_b b x_b$ and $x_c c x_c$ are complete factors but b, c, $x_b \bar{b} x_b$, $x_c \bar{c} x_c$, x_b^j and x_c^j for $j > 1$ are not, P encodes the assignment to the inputs of C_i that makes them true or false according to τ.

Since these are all the possibilities for how P can end, each diverse palindromic factorization of S_i encodes some assignment to the inputs of C_i. This gives us the following lemma:

Lemma 15. *If we have a string S_{i-1} that represents C_{i-1} and C_i is obtained from C_{i-1} by splitting an output of C_{i-1} into two outputs, then in constant time we can append symbols to S_{i-1} to obtain a string S_i that represents C_i.*

5.4 Adding a NAND gate

Finally, suppose C_i is obtained from C_{i-1} by making two outputs of C_{i-1} labelled a and b the inputs of a new NAND gate whose output is labelled c. Let C'_{i-1} be the circuit obtained from C_{i-1} by splitting the output of C_{i-1} labelled a into two outputs labelled a_1 and a_2, where a_1 and a_2 are symbols we use only here. Assuming S_{i-1} represents C_{i-1}, we can use Lemma 15 to build in constant time a string S'_{i-1} representing C'_{i-1}. We set

$$\begin{aligned} S'_i \;=\; & S'_{i-1}\,\$\#\, x_{c'}^3 a'_1 x_{c'} a_1 x_{c'} \overline{a_1} x_{c'} \overline{a'_1} x_{c'}^5 \\ & \$'\#'\, x_{c'}^7 a'_2 x_{c'} a_2 x_{c'} \overline{a_2} x_{c'} \overline{a'_2} x_{c'}^9 \\ & \$''\#''\, x_{c'}^{11} b' x_{c'} b x_{c'} \bar{b} x_{c'} \overline{b'} x_{c'}^{13}\,, \end{aligned}$$

where all of the symbols in the suffix after S'_{i-1} are ones we use only here.

Since \$, \$′, \$″, \$‴, # and #′ do not occur in S_{i-1} and occur as pairs of consecutive characters in S'_i, they must each be complete factors in any palindromic factorization of S'_i. Therefore, any diverse palindromic factorization of S'_i consists of

1. a diverse palindromic factorization of S'_{i-1},
2. \$, #,
3. a diverse palindromic factorization of $x_{c'}^3 a'_1 x_{c'} a_1 x_{c'} \overline{a_1} x_{c'} \overline{a'_1} x_{c'}^5$,
4. \$′, #′,
5. a diverse palindromic factorization of $x_{c'}^7 a'_2 x_{c'} a_2 x_{c'} \overline{a_2} x_{c'} \overline{a'_2} x_{c'}^9$,
6. \$″, #″,
7. a diverse palindromic factorization of $x_{c'}^{11} b' x_{c'} b x_{c'} \bar{b} x_{c'} \overline{b'} x_{c'}^{13}$.

If a_1 is a complete factor in the factorization of S'_{i-1}, then the diverse palindromic factorization of

$$x_{c'}^3 a'_1 x_{c'} a_1 x_{c'} \overline{a_1} x_{c'} \overline{a'_1} x_{c'}^5$$

must include either

$$a'_1,\ x_{c'} a_1 x_{c'},\ \overline{a_1},\ x_{c'} \overline{a'_1} x_{c'} \qquad \text{or} \qquad a'_1,\ x_{c'} a_1 x_{c'},\ \overline{a_1},\ x_{c'},\ \overline{a'_1}\,.$$

Notice that in the former case, the factorization need not contain $x_{c'}$. If $\overline{a_1}$ is a complete factor in the factorization of S'_{i-1}, then the diverse palindromic factorization of

$$x_{c'}^3 a'_1 x_{c'} a_1 x_{c'} \overline{a_1} x_{c'} \overline{a'_1} x_{c'}^5$$

must include either

$$x_{c'} a'_1 x_{c'},\ a_1,\ x_{c'} \overline{a_1} x_{c'},\ \overline{a'_1} \qquad \text{or} \qquad a'_1,\ x_{c'},\ a_1,\ x_{c'} \overline{a_1} x_{c'},\ \overline{a'_1}\,.$$

Again, in the former case, the factorization need not contain $x_{c'}$. Symmetric propositions hold for a_2 and b.

We set

$$S''_i = S'_i\, \$^\dagger \#^\dagger\, x_{c'}^{15} \overline{a'_1} x_{c'} c' x_{c'} \overline{b'} x_{c'}^{17}\, \$^{\dagger\dagger} \#^{\dagger\dagger}\, x_{c'}^{19} \overline{a'_2} x_{c'} d x_{c'} b' x_{c'}^{21}\,,$$

where $\$^\dagger$, $\#^\dagger$, $\$^{\dagger\dagger}$, $\#^{\dagger\dagger}$, c' and d are symbols we use only here. Any diverse palindromic factorization of S_i'' consists of

1. a diverse palindromic factorization of S_i',
2. $\$^\dagger,\ \#^\dagger$,
3. a diverse palindromic factorization of $x_{c'}^{15}\overline{a_1'}x_{c'}c'x_{c'}\overline{b'}x_{c'}^{17}$,
4. $\$^{\dagger\dagger},\ \#^{\dagger\dagger}$,
5. a diverse palindromic factorization of $x_{c'}^{19}\overline{a_2'}x_{c'}dx_{c'}b'x_{c'}^{21}$.

Since a_1 and a_2 label outputs in C_{i-1}' split from the same output in C_{i-1}, it follows that a_1 is a complete factor in a diverse palindromic factorization of S_{i-1}' if and only if a_2 is. Therefore, we need consider only four cases:

Case 1: The factorization of S_{i-1}' includes a_1, a_2 and b as complete factors, so the factorization of S_i' includes as complete factors either $x_{c'}\overline{a_1'}x_{c'}$, or $\overline{a_1'}$ and $x_{c'}$; either $x_{c'}\overline{a_2'}x_{c'}$, or $\overline{a_2'}$ and $x_{c'}$; either $x_{c'}\overline{b'}x_{c'}$, or $\overline{b'}$ and $x_{c'}$; and b'. Trying all the combinations — there are only four, since $x_{c'}$ can appear as a complete factor at most once — shows that any diverse palindromic factorization of S_i'' includes one of

$$\overline{a_1'},\ x_{c'}c'x_{c'},\ \overline{b'},\ \ldots,\ \overline{a_2'},\ x_{c'},\ d,\ x_{c'}b'x_{c'}\ ,$$
$$\overline{a_1'},\ x_{c'}c'x_{c'},\ \overline{b'},\ \ldots,\ \ x_{c'}\overline{a_2'}x_{c'},\ d,\ x_{c'}b'x_{c'}\ ,$$

with the latter only possible if $x_{c'}$ appears earlier in the factorization.

Case 2: The factorization of S_{i-1}' includes a_1, a_2 and $\overline{b}$ as complete factors, so the factorization of S_i' includes as complete factors either $x_{c'}\overline{a_1'}x_{c'}$, or $\overline{a_1'}$ and $x_{c'}$; either $x_{c'}\overline{a_2'}x_{c'}$, or $\overline{a_2'}$ and $x_{c'}$; $\overline{b'}$; and either $x_{c'}b'x_{c'}$, or b' and $x_{c'}$. Trying all the combinations shows that any diverse palindromic factorization of S_i'' includes one of

$$\overline{a_1'},\ x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ \ldots,\ \overline{a_2'},\ x_{c'}dx_{c'},\ b'\ ,$$
$$x_{c'}\overline{a_1'}x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ \ldots,\ \overline{a_2'},\ x_{c'}dx_{c'},\ b'\ ,$$

with the latter only possible if $x_{c'}$ appears earlier in the factorization.

Case 3: The factorization of S'_{i-1} includes $\overline{a_1}$, $\overline{a_2}$ and b as complete factors, so the factorization of S'_i includes as complete factors $\overline{a'_1}$; $\overline{a'_2}$; either $x_{c'}\overline{b'}x_{c'}$, or $\overline{b'}$ and $x_{c'}$; and b'. Trying all the combinations shows that any diverse palindromic factorization of S''_i includes one of

$$x_{c'}\overline{a'_1}x_{c'},\ c',\ x_{c'},\ \overline{b'},\ \ldots,\ x_{c'}\overline{a'_2}x_{c'},\ d,\ x_{c'}b'x_{c'}\ ,$$
$$x_{c'}\overline{a'_1}x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ \ldots,\ x_{c'}\overline{a'_2}x_{c'},\ d,\ x_{c'}b'x_{c'}\ ,$$

with the latter only possible if $x_{c'}$ appears earlier in the factorization.

Case 4: The factorization of S'_{i-1} includes $\overline{a_1}$, $\overline{a_2}$ and $\overline{b}$ as complete factors, so the factorization of S'_i includes as complete factors $\overline{a'_1}$; $\overline{a'_2}$; $\overline{b'}$; and either $x_{c'}b'x_{c'}$, or b' and $x_{c'}$. Trying all the combinations shows that any diverse palindromic factorization of S''_i that extends the factorization of S'_i includes one of

$$x_{c'}\overline{a'_1}x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ \ldots,\ x_{c'}\overline{a'_2}x_{c'},\ d,\ x_{c'},\ b'\ ,$$
$$x_{c'}\overline{a'_1}x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ \ldots,\ x_{c'}\overline{a'_2}x_{c'},\ d,\ x_{c'}b'x_{c'}\ ,$$

with the latter only possible if $x_{c'}$ appears earlier in the factorization.

Summing up, any diverse palindromic factorization of S''_i always includes $x_{c'}$ and includes either $x_{c'}c'x_{c'}$ if the factorization of S'_{i-1} includes a_1, a_2 and b as complete factors, or c' otherwise.

We set

$$S'''_i = S''_i\,\$^{\dagger\dagger\dagger}\#^{\dagger\dagger\dagger}\,x_{c'}^{23}c''x_{c'}c'x_{c'}\overline{c'}x_{c'}\overline{c''}x_{c'}^{25}\ ,$$

where $\$^{\dagger\dagger\dagger}$ and $\#^{\dagger\dagger\dagger}$ are symbols we use only here. Any diverse palindromic factorization of S'''_i consists of

1. a diverse palindromic factorization of S''_i,
2. $\$^{\dagger\dagger\dagger}$, $\#^{\dagger\dagger\dagger}$,
3. a diverse palindromic factorization of $x_{c'}^{23}c''x_{c'}c'x_{c'}\overline{c'}x_{c'}\overline{c''}x_{c'}^{25}$.

Since $x_{c'}$ must appear as a complete factor in the factorization of S''_i, if c' is a complete factor in the factorization of S''_i, then the factorization of

$$x_{c'}^{23}c''x_{c'}c'x_{c'}\overline{c'}x_{c'}\overline{c''}x_{c'}^{25}$$

must include

$$c'',\ x_{c'}c'x_{c'},\ \overline{c'},\ x_{c'}\overline{c''}x_{c'}\,;$$

otherwise, it must include

$$x_{c'}c''x_{c'},\ c',\ x_{c'}\overline{c'}x_{c'},\ \overline{c''}\,.$$

That is, the factorization of $x_{c'}^{23}c''x_{c'}c'x_{c'}\overline{c'}x_{c'}\overline{c''}x_{c'}^{25}$ includes c'', $x_{c'}$ and $x_{c'}\overline{c''}x_{c'}$ but not $\overline{c''}$ or $x_{c'}c''x_{c'}$, if and only if the factorization of S_i'' includes c'; otherwise, it includes $\overline{c''}$, $x_{c'}$ and $x_{c'}c''x_{c'}$ but not c'' or $x_{c'}\overline{c''}x_{c'}$.

We set

$$S_i = S_i'''\,\$^{\ddagger}\#^{\ddagger}\,x_ccx_cc''x_c\overline{c''}x_c\bar{c}x_c\,,$$

where $\$^{\ddagger}$, $\#^{\ddagger}$, c, $\bar{c}$ and x_c are symbols that do not appear in S_i'''. Any diverse palindromic factorization of S_i consists of

1. a diverse palindromic factorization of S_i''',
2. $\$^{\ddagger}$, $\#^{\ddagger}$,
3. a diverse palindromic factorization of $x_ccx_cc''x_c\overline{c''}x_c\bar{c}x_c$.

Since exactly one of c'' and $\overline{c''}$ must appear as a complete factor in the factorization of S_i''', the factorization of

$$x_ccx_cc''x_c\overline{c''}x_c\bar{c}x_c$$

must be either

$$x_c,\ c,\ x_cc''x_c,\ \overline{c''},\ x_c\bar{c}x_c$$

or

$$x_ccx_c,\ c'',\ x_c\overline{c''}x_c,\ \bar{c},\ x_c\,.$$

Thus if c'' is a complete factor in the factorization of S_i''', then c, x_c and $x_c\bar{c}x_c$ are complete factors in the factorization of S_i but $\bar{c}$, x_ccx_c and x_c^j are not for $j > 1$; otherwise, $\bar{c}$, x_c and x_ccx_c are complete factors but c, $x_c\bar{c}x_c$ and x_c^j are not for $j > 1$.

Assume S_{i-1} represents C_{i-1}. Let τ be an assignment to the inputs of C_{i-1} and let P be a diverse palindromic factorization of S_{i-1} encoding τ. By Lemma 15 we can extend P to P' so that it encodes the assignment to the inputs of C_{i-1}' that makes them true or false according to τ. There are four cases to consider:

Case 1: τ makes the outputs of C_{i-1} labelled a and b both true. Then P' concatenated with, e.g.,

$$\$,\ \#,\ x_{c'}^3,\ a_1',\ x_{c'}a_1x_{c'},\ \overline{a_1},\ x_{c'}\overline{a_1'}x_{c'},\ x_{c'}^4,$$
$$\$',\ \#',\ x_{c'}^7,\ a_2',\ x_{c'}a_2x_{c'},\ \overline{a_2},\ x_{c'}\overline{a_2'}x_{c'},\ x_{c'}^8,$$
$$\$'',\ \#'',\ x_{c'}^{11},\ b',\ x_{c'}bx_{c'},\ \bar{b},\ x_{c'}\overline{b'}x_{c'},\ x_{c'}^{12}$$

is a diverse palindromic factorization P'' of S_i' which, concatenated with, e.g.,

$$\$^\dagger,\ \#^\dagger,\ x_{c'}^{15},\ \overline{a_1'},\ x_{c'}c'x_{c'},\ \overline{b'},\ x_{c'}^{17},$$
$$\$^{\dagger\dagger},\ \#^{\dagger\dagger},\ x_{c'}^{19},\ \overline{a_2'},\ x_{c'},\ d,\ x_{c'}b'x_{c'},\ x_{c'}^{20}$$

is a diverse palindromic factorization P''' of S_i'' which, concatenated with, e.g.,

$$\$^{\dagger\dagger\dagger},\ \#^{\dagger\dagger\dagger},\ x_{c'}^{22},\ x_{c'}c''x_{c'},\ c',\ x_{c'}\overline{c'}x_{c'},\ \overline{c''},\ x_{c'}^{25}$$

is a diverse palindromic factorization $P^\dagger$ of S_i''' which, concatenated with

$$\$^\ddagger,\ \#^\ddagger,\ x_cx_c,\ c'',\ x_c\overline{c''}x_c,\ \bar{c},\ x_c$$

is a diverse palindromic factorization $P^\ddagger$ of S_i in which $\bar{c}$, x_c and x_cx_c are complete factors but c, $x_c\bar{c}x_c$ and x_c^j are not for $j > 1$.

Case 2: τ makes the output of C_{i-1} labelled a true but the output labelled b false. Then P' concatenated with, e.g.,

$$\$,\ \#,\ x_{c'}^3,\ a_1',\ x_{c'}a_1x_{c'},\ \overline{a_1},\ x_{c'}\overline{a_1'}x_{c'},\ x_{c'}^4,$$
$$\$',\ \#',\ x_{c'}^7,\ a_2',\ x_{c'}a_2x_{c'},\ \overline{a_2},\ x_{c'}\overline{a_2'}x_{c'},\ x_{c'}^8,$$
$$\$'',\ \#'',\ x_{c'}^{10},\ x_{c'}b'x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'},\ x_{c'}^{13}$$

is a diverse palindromic factorization P'' of S_i' which, concatenated with, e.g.,

$$\$^\dagger,\ \#^\dagger,\ x_{c'}^{15},\ \overline{a_1'},\ x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ x_{c'}^{16},$$
$$\$^{\dagger\dagger},\ \#^{\dagger\dagger},\ x_{c'}^{19},\ \overline{a_2'},\ x_{c'}dx_{c'},\ b',\ x_{c'}^{21}$$

is a diverse palindromic factorization P''' of S_i'' which, concatenated with, e.g.,

$$\$^{\dagger\dagger\dagger},\ \#^{\dagger\dagger\dagger},\ x_{c'}^{23},\ c'',\ x_{c'}c'x_{c'},\ \overline{c'},\ x_{c'}\overline{c''}x_{c'},\ x_{c'}^{24}$$

is a diverse palindromic factorization $P^{\dagger}$ of S_i''' which, concatenated with

$$\$^{\ddagger},\ \#^{\ddagger},\ x_c,\ c,\ x_cc''x_c,\ \overline{c''},\ x_c\bar{c}x_c$$

is a diverse palindromic factorization $P^{\ddagger}$ of S_i in which c, $x_c\bar{c}x_c$ and x_c are complete factors but $\bar{c}$, x_ccx_c and x_c^j are not for $j > 1$.

Case 3: τ makes the output of C_{i-1} labelled a false but the output labelled b true. Then P' concatenated with, e.g.,

$$\begin{gathered}\$,\ \#,\ x_{c'}^{2},\ x_{c'}a_1'x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a_1'},\ x_{c'}^{5},\\ \$',\ \#',\ x_{c'}^{6},\ x_{c'}a_2'x_{c'},\ a_2,\ x_{c'}\overline{a_2}x_{c'},\ \overline{a_2'},\ x_{c'}^{9},\\ \$'',\ \#'',\ x_{c'}^{11},\ b',\ x_{c'}bx_{c'},\ \bar{b},\ x_{c'}\overline{b'}x_{c'},\ x_{c'}^{12}\end{gathered}$$

is a diverse palindromic factorization P'' of S_i' which, concatenated with, e.g.,

$$\begin{gathered}\$^{\dagger},\ \#^{\dagger},\ x_{c'}^{14},\ x_{c'}\overline{a_1'}x_{c'},\ c',\ x_{c'},\ \overline{b'},\ x_{c'}^{17},\\ \$^{\dagger\dagger},\ \#^{\dagger\dagger},\ x_{c'}^{18},\ x_{c'}\overline{a_2'}x_{c'},\ d,\ x_{c'}b'x_{c'},\ x_{c'}^{20}\end{gathered}$$

is a diverse palindromic factorization P''' of S_i'' which, concatenated with, e.g.,

$$\$^{\dagger\dagger\dagger},\ \#^{\dagger\dagger\dagger},\ x_{c'}^{23},\ c'',\ x_{c'}c'x_{c'},\ \overline{c'},\ x_{c'}\overline{c''}x_{c'},\ x_{c'}^{24}$$

is a diverse palindromic factorization $P^{\dagger}$ of S_i''' which, concatenated with

$$\$^{\ddagger},\ \#^{\ddagger},\ x_c,\ c,\ x_cc''x_c,\ \overline{c''},\ x_c\bar{c}x_c$$

is a diverse palindromic factorization $P^{\ddagger}$ of S_i in which c, $x_c\bar{c}x_c$ and x_c are complete factors but $\bar{c}$, x_ccx_c and x_c^j are not for $j > 1$.

Case 4: τ makes the outputs of C_{i-1} labelled a and b both false. Then P' concatenated with,

e.g.,

$$\$,\ \#,\ x_{c'}^2,\ x_{c'}a_1'x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a_1'},\ x_{c'}^5,$$
$$\$',\ \#',\ x_{c'}^6,\ x_{c'}a_2'x_{c'},\ a_2,\ x_{c'}\overline{a_2}x_{c'},\ \overline{a_2'},\ x_{c'}^9,$$
$$\$'',\ \#'',\ x_{c'}^{10},\ x_{c'}b'x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'},\ x_{c'}^{13}$$

is a diverse palindromic factorization P'' of S_i' which, concatenated with, e.g.,

$$\$^\dagger,\ \#^\dagger,\ x_{c'}^{14},\ x_{c'}\overline{a_1'}x_{c'},\ c',\ x_{c'}\overline{b'}x_{c'},\ x_{c'}^{16},$$
$$\$^{\dagger\dagger},\ \#^{\dagger\dagger},\ x_{c'}^{18},\ x_{c'}\overline{a_2'}x_{c'},\ d,\ x_{c'},\ b',\ x_{c'}^{21}$$

is a diverse palindromic factorization P''' of S_i'' which, concatenated with, e.g.,

$$\$^{\dagger\dagger\dagger},\ \#^{\dagger\dagger\dagger},\ x_{c'}^{23},\ c'',\ x_{c'}c'x_{c'},\ \overline{c'},\ x_{c'}\overline{c''}x_{c'},\ x_{c'}^{24}$$

is a diverse palindromic factorization $P^\dagger$ of S_i''' which, concatenated with

$$\$^\ddagger,\ \#^\ddagger,\ x_c,\ c,\ x_cc''x_c,\ \overline{c''},\ x_c\bar{c}x_c$$

is a diverse palindromic factorization $P^\ddagger$ of S_i in which c, $x_c\bar{c}x_c$ and x_c are complete factors but $\bar{c}$, x_ccx_c and x_c^j are not for $j > 1$.

Notice that in all cases $P^\ddagger$ encodes the assignment to the inputs of C_i that makes them true or false according to τ. Since C_{i-1} and C_i have the same inputs, each assignment to the inputs of C_i is encoded by some diverse palindromic factorization of S_i.

Now let P be a diverse palindromic factorization of S_i and let τ be the assignment to the inputs of C_{i-1} that is encoded by a prefix of P. Let $\hat{P}$ be a diverse palindromic factorization of S_{i-1}'. Since a_1 and a_2 are obtained by splitting a in S_{i-1}, it follows that a_1 is a complete factor of $\hat{P}$ if and only if a_2 is. Therefore, in what follows we only consider any diverse palindromic factorization P of S_i in which either both a_1 and a_2 are complete factors, or neither a_1 nor a_2 is a complete factor.

Let P' be the prefix of P that is a diverse palindromic factorization of S_i'''.

Case A: Suppose the factorization of

$$x_{c'}^{23}c''x_{c'}c'x_{c'}\overline{c'}x_{c'}\overline{c''}x_{c'}^{25}$$

in P' includes $\overline{c''}$ as a complete factor, which is the case if and only if P includes $\bar{c}$, x_c and $x_c c x_c$ as complete factors but not c, $x_c \bar{c} x_c$ and x_c^j for $j > 1$. We will show that τ must make the outputs of C_{i-1} labelled a and b true. Let P'' be the prefix of P' that is a diverse palindromic factorization of S_i''. Since $\overline{c''}$ is a complete factor in the factorization of

$$x_{c'}^{23} c'' x_{c'} c' x_{c'} \overline{c'} x_{c'} \overline{c''} x_{c'}^{25}$$

in P', so is c'. Therefore, c' is not a complete factor in the factorization of

$$x_{c'}^{15} \overline{a_1'} x_{c'} c' x_{c'} \overline{b'} x_{c'}^{17}$$

in P'', so $\overline{a_1'}$ and $\overline{b'}$ are.

Let P''' be the prefix of P'' that is a diverse palindromic factorization of S_i'. Since $\overline{a_1'}$ and $\overline{b'}$ are complete factors later in P'', they are not complete factors in P'''. Therefore, $\overline{a_1}$ and $\bar{b}$ are complete factors in the factorizations of

$$x_{c'}^{3} a_1' x_{c'} a_1 x_{c'} \overline{a_1} x_{c'} \overline{a_1'} x_{c'}^{5} \qquad \text{and} \qquad x_{c'}^{11} b' x_{c'} b x_{c'} \bar{b} x_{c'} \overline{b'} x_{c'}^{13}$$

in P''', so they are not complete factors in the prefix $P^\dagger$ of P that is a diverse palindromic factorization of S_{i-1}'. Since we built S_{i-1}' from S_{i-1} with Lemma 15, it follows that a_1 and b are complete factors in the prefix of P that encodes τ. Therefore, τ makes the outputs of C_{i-1} labelled a and b true.

Case B: Suppose the factorization of

$$x_{c'}^{23} c'' x_{c'} c' x_{c'} \overline{c'} x_{c'} \overline{c''} x_{c'}^{25}$$

in P' does not include $\overline{c''}$ as a complete factor, which implies that it does include $x_{c'} \overline{c''} x_{c'}$ as a complete factor. Since, as noted earlier, we can assume that a_1 is a complete factor of P if and only if a_2 is, it follows that the factorization of

$$x_{c'}^{23} c'' x_{c'} c' x_{c'} \overline{c'} x_{c'} \overline{c''} x_{c'}^{25}$$

must include

$$c'', x_{c'} c' x_{c'}, \overline{c'}, x_{c'} \overline{c''} x_{c'}.$$

Then, P must include x_c, c and $\overline{c''}$ as complete factors. We will show that τ must make at least

one of the outputs of C_{i-1} labelled a or b false. Let P'' be the prefix of P' that is a diverse palindromic factorization of S_i''. Since $x_{c'}c'x_{c'}$ is a complete factor in the factorization of

$$x_{c'}^{23}c''x_{c'}c'x_{c'}\overline{c'}x_{c'}\overline{c''}x_{c'}^{25}$$

in P', c' is a complete factor in the factorization of

$$x_{c'}^{15}\overline{a_1'}x_{c'}c'x_{c'}\overline{b'}x_{c'}^{17}$$

in P''. Then, the factorization of

$$x_{c'}^{15}\overline{a_1'}x_{c'}c'x_{c'}\overline{b'}x_{c'}^{17}$$

must include one of the following three:

$$x_{c'}\overline{a_1'}x_{c'}, c', x_{c'}\overline{b'}x_{c'}, \tag{5.1}$$

$$x_{c'}\overline{a_1'}x_{c'}, c', x_{c'}, \overline{b'}, \tag{5.2}$$

$$\overline{a_1'}, x_{c'}, c', x_{c'}\overline{b'}x_{c'}. \tag{5.3}$$

Case B-a: Assume the factorization of $x_{c'}^{15}\overline{a_1'}x_{c'}c'x_{c'}\overline{b'}x_{c'}^{17}$ includes (5.1). Let P''' be the prefix of P'' that is a diverse palindromic factorization of S_i'. Since $\overline{a_1'}$ and $\overline{b'}$ are not complete factors later in P'', they are complete factors in P'''. Therefore, there are five combinations of factorizations of

$$x_{c'}^{3}a_1'x_{c'}a_1x_{c'}\overline{a_1}x_{c'}\overline{a_1'}x_{c'}^{5} \qquad \text{and} \qquad x_{c'}^{11}b'x_{c'}bx_{c'}\bar{b}x_{c'}\overline{b'}x_{c'}^{13}$$

in P''', as follows:

Case B-a1: The factorizations include

$$x_{c'}a_1'x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a_1'} \text{ and } x_{c'}b'x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'}.$$

In this case, a_1 and b are not complete factors in the prefix of P that encodes τ. Therefore, τ makes both the outputs of C_{i-1} labelled a and b false.

Case B-a2: The factorizations include

$$x_{c'}a_1'x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a_1'} \text{ and } b',\ x_{c'}bx_{c'},\ \bar{b},\ x_{c'},\ \overline{b'}.$$

In this case, a_1 is not a complete factor and b is a complete factor in the prefix of P that encodes τ. Therefore, τ makes the outputs of C_{i-1} labelled a false and b true.

Case B-a3: The factorizations include

$$a'_1,\ x_{c'}a_1x_{c'},\ \overline{a_1},\ x_{c'},\ \overline{a'_1} \text{ and } x_{c'}b'x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'}.$$

In this case, a_1 is a complete factor and b is not a complete factor in the prefix of P that encodes τ. Therefore, τ makes the outputs of C_{i-1} labelled a true and b false.

Case B-a4: The factorizations include

$$a'_1,\ x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a'_1} \text{ and } x_{c'}b'x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'}.$$

In this case, a_1 and b are not complete factors in the prefix of P that encodes τ. Therefore, τ makes both the outputs of C_{i-1} labelled a and b false.

Case B-a5: The factorizations include

$$x_{c'}a'_1x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a'_1} \text{ and } b',\ x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'}.$$

In this case, a_1 and b are not complete factors in the prefix of P that encodes τ. Therefore, τ makes both the outputs of C_{i-1} labelled a and b false.

Case B-b: Assume the factorization of $x_{c'}^{15}\overline{a'_1}x_{c'}c'x_{c'}\overline{b'}x_{c'}^{17}$ includes (5.2). Let P'' be the prefix of P' that is a diverse palindromic factorization of S''_i. Let P''' be the prefix of P'' that is a diverse palindromic factorization of S'_i. Since $\overline{a'_1}$ and $x_{c'}\overline{b'}x_{c'}$ are not complete factors later in P'', they are complete factors in P'''. Therefore, the factorizations of

$$x_{c'}^3a'_1x_{c'}a_1x_{c'}\overline{a_1}x_{c'}\overline{a'_1}x_{c'}^5 \qquad \text{and} \qquad x_{c'}^{11}b'x_{c'}bx_{c'}\bar{b}x_{c'}\overline{b'}x_{c'}^{13}$$

must include

$$x_{c'}a'_1x_{c'},\ a_1,\ x_{c'}\overline{a_1}x_{c'},\ \overline{a'_1} \text{ and } b',\ x_{c'}bx_{c'},\ \bar{b},\ x_{c'}\overline{b'}x_{c'}$$

in P'''. Then a_1 is not a complete factor and b is a complete factor in the prefix of P that encodes τ. Therefore, τ makes the outputs of C_{i-1} labelled a false and b true.

Case B-c: Assume the factorization of $x_{c'}^{15}\overline{a'_1}x_{c'}c'x_{c'}\overline{b'}x_{c'}^{17}$ includes (5.3). Let P'' be the prefix of P' that is a diverse palindromic factorization of S''_i. Let P''' be the prefix of P'' that is

a diverse palindromic factorization of S'_i. Since $x_{c'}\overline{a'_1}x_{c'}$ and $\overline{b'}$ are not complete factors later in P'', they are complete factors in P'''. Therefore, the factorizations of

$$x_{c'}^3 a'_1 x_{c'} a_1 x_{c'} \overline{a_1} x_{c'} \overline{a'_1} x_{c'}^5 \qquad \text{and} \qquad x_{c'}^{11} b' x_{c'} b x_{c'} \bar{b} x_{c'} \overline{b'} x_{c'}^{13}$$

must include

$$a'_1,\ x_{c'}a_1x_{c'},\ \overline{a_1},\ x_{c'}\overline{a'_1}x_{c'} \text{ and } x_{c'}b'x_{c'},\ b,\ x_{c'}\bar{b}x_{c'},\ \overline{b'}$$

in P'''. Then a_1 is a complete factor and b is not a complete factor in the prefix of P that encodes τ. Therefore, τ makes the outputs of C_{i-1} labelled a true and b false.

The above arguments give the following lemma.

Lemma 16. *If we have a string S_{i-1} that represents C_{i-1} and C_i is obtained from C_{i-1} by making two outputs of C_{i-1} the inputs of a new NAND gate, then in constant time we can append symbols to S_{i-1} to obtain a string S_i that represents C_i.*

5.5 Summing up

By Lemmas 14, 15 and 16 and induction, given a Boolean circuit C composed only of splitters and NAND gates with two inputs and one output, in time linear in the size of C we can build, inductively and in turn, a sequence of strings $S_1, \ldots, S_t$ such that S_i represents C_i. As mentioned in introduction , once we have S_t we can easily build in constant time a string S that has a diverse palindromic factorization if and only if C is satisfiable. Therefore, diverse palindromic factorization is NP-hard. Since it is obviously in NP, we have the following theorem:

Theorem 9. *Diverse palindromic factorization is NP-complete.*

5.6 k-Diverse factorization

It is not difficult to check that our reduction is still correct even if factors of the forms \$, $\#$ and x^j for $j > 1$ can appear arbitrarily often in the factorization, as long as factors of the forms a, x and xax can each appear at most once. (By "of the form" we mean equal up to subscripts, bars and superscripts apart from exponents; a stands for any letter except x.) It follows that it is still NP-complete to decide for any fixed k whether a string can be factored into palindromes that each appear at most k times in the factorization.

Suppose we are given k and a Boolean circuit C composed only of splitters and NAND gates with two inputs and one output. In linear time we can build, as we have described, a string S such that S has a diverse palindromic factorization if and only if C is satisfiable. In linear time we can then build a string T as follows: we start with T equal to the empty string; for each substring of S of the form a, we append to T a substring of the form

$$\$_1\#_1\, a\, \$_2\#_2\, a\, \$_3\#_3 \cdots \$_{k-1}\#_{k-1}\, a\, \$_k\#_k\,,$$

where $\$_1, \ldots, \$_k, \#_1, \ldots, \#_k$ are symbols we use only here; for each substring of S of the form x, we append to T a substring of the form

$$\$'_1\#'_1\, x\, \$'_2\#'_2\, x\, \$'_3\#'_3 \cdots \$'_{k-1}\#'_{k-1}\, x\, \$'_k\#'_k\,,$$

where $\$'_1, \ldots, \$'_k, \#'_1, \ldots, \#'_k$ are symbols we use only here; for each substring of S of the form xax, we append to T a substring of the form

$$\$''_1\#''_1\, xax\, \$''_2\#''_2\, xax\, \$''_3\#''_3 \cdots \$''_{k-1}\#''_{k-1}\, xax\, \$''_k\#''_k\,,$$

where $\$''_1, \ldots, \$''_k, \#''_1, \ldots, \#''_k$ are symbols we use only here.

Notice that the only k-diverse palindromic factorization of T includes each substring of S of the forms a, x and xax exactly $k-1$ times each. In particular, any substring of T of the form xax cannot be factored into $x,\ a,\ x$, because x must appear $k-1$ times elsewhere in the factorization. Therefore, there is a k-diverse palindromic factorization of $S\,\$\#\,T$, where $\$$ and $\#$ are symbols we use only here, if and only if there is a diverse palindromic factorization of S and, thus, if and only if C is satisfiable. This implies the following generalization of Theorem 9.

Theorem 10. *For any fixed $k \geq 1$, k-diverse palindromic factorization is NP-complete.*

5.7 Binary alphabet

The reduction described above involves multiple distinct symbols for each component of the circuit and thus requires an unbounded alphabet, but we will next show that a binary alphabet is sufficient.

Let S be an arbitrary string and let Σ be the set of distinct symbols occurring in S. Let δ be an (arbitrary) bijective mapping $\delta : \Sigma \to \{ba^ib : i \in [1..|\Sigma|]\}$. We will also use δ to denote

the implied mapping from Σ^* to $\{a, b\}^*$ defined recursively by $\delta(X\alpha) = \delta(X) \cdot \delta(\alpha)$ for any $X \in \Sigma^*$ and $\alpha \in \Sigma$.

Notice that δ preserves palindromes, i.e., for any palindrome $P \in \Sigma^*$, $\delta(P)$ is a palindrome too. Thus, if $\mathbf{P} = (P_1, P_2, \ldots, P_k)$ is a palindromic factorization of S, then $\delta(\mathbf{P}) = (\delta(P_1), \delta(P_2), \ldots, \delta(P_k))$ is a palindromic factorization of $\delta(S)$. Furthermore any palindrome in $\delta(S)$ of the form $(ba^+b)^+$ must be a preserved palindrome, i.e., an image $\delta(P)$ of a palindrome P occurring in S. Any palindromic factorization of $\delta(S)$ consisting of preserved palindromes only corresponds to a palindromic factorization of S. We call this a preserved palindromic factorization of $\delta(S)$. Notice that a preserved palindromic factorization $\delta(\mathbf{P})$ is diverse if and only if $\mathbf{P}$ is diverse.

Now consider an arbitrary non-preserved palindromic factorization of $\delta(S)$. It is easy to see that the first palindrome must be either a single b or a preserved palindrome. Furthermore, any palindrome following a preserved palindrome in the factorization must be either a single b or a preserved palindrome. Thus the palindromic factorization of $\delta(S)$ begins with a (possibly empty) sequence of preserved palindromes followed by a single b. A symmetric argument shows that the factorization also ends with a (possibly empty) sequence of preserved palindromes preceded by a single b. The two single b's cannot be the same b since one is the first b in an image of a symbol in S, and the other is a last b. Thus a non-preserved palindromic factorization can never be diverse.

The above discussion proves the following lemma.

Lemma 17. *For any string S, $\delta(S)$ has a diverse palindromic factorization if and only if S has a diverse palindromic factorization.*

Applying the lemma to the string S constructed from a Boolean circuit C as described in Sections 5.2, 5.3 and 5.4, shows that $\delta(S)$ has a diverse palindromic factorization if and only if C is satisfiable. Since $\delta(S)$ can be constructed in time quadratic in the size of C, we have a binary alphabet version of Theorem 9.

Theorem 11. *Diverse palindromic factorization of binary strings is NP-complete.*

If we allow each factor to occur at most $k > 1$ times, the above transformation to a binary alphabet does not work anymore, because two single b's is now allowed. However, a small modification is sufficient to correct this. First, we replace δ with a bijection $\delta' : \Sigma \to \{ba^ib : i \in [3..|\Sigma| + 2]\}$. Second, we append to $\delta'(S)$ the string Q_k which is a length $20k$ prefix of $(abbaab)^*$.

Let us first analyze the palindromic structure of Q_k. It is easy to see that the only palindromes in Q_k are

$$a,\ b,\ aa,\ bb,\ aba,\ bab,\ abba, \text{ and } baab.$$

The total length of these palindromes is 20 and thus the only possible k-diverse palindromic factorization of Q_k is one where all the above palindromes appear exactly k times. Such factorizations exist too. For example, k copies of

$$abba,\ aba,\ bb,\ aa,\ bab,\ baab$$

followed by $2k$ single symbol palindromes is such a factorization.

Now consider the string $\delta'(S)Q_k$. It is easy to verify that the only palindromes overlapping both $\delta'(S)$ and Q_k are aba and bab. However, in any palindromic factorization containing one of them, the factorization of the remaining part of Q_k together with the overlapping palindrome would have to contain more than k occurrences of some factor. Thus in any k-diverse palindromic factorization of $\delta'(S)Q_k$, there are no overlapping palindromes and the factorizations of $\delta'(S)$ and Q_k are separate. Since the factorization of Q_k contains k single b's, the factorization of $\delta'(S)$ cannot contain any single b's. Then, by the discussion earlier in this section, all palindromes in $\delta'(S)$ must be preserved palindromes.

Lemma 18. *For any string S and any $k \geq 1$, the string $\delta'(S)Q_k$ has a k-diverse palindromic factorization if and only if S has a k-diverse palindromic factorization.*

Combining this with Theorem 10, we obtain the following:

Theorem 12. *For any fixed $k \geq 1$, k-diverse palindromic factorization of binary strings is NP-complete.*

Chapter 6

Closed Factorization

In this chapter, we introduce a new string factorization problem, called the closed factorization problem, and present a linear time algorithm that computes the closed factorization of a given string.

The results in this chapter were originally published in [8].

6.1 Closed strings and closed factorization

A string X is said to be *closed*, if there exists a border b of X that occurs exactly twice in X, i.e., $b = X[1..|b|] = X[|X| - |b| + 1..|X|]$ and $b \neq X[i..i + |b| - 1]$ for any $2 \leq i \leq |X| - |b|$. We suppose that any single character $c \in \Sigma$ is closed, assuming that the empty string ε occurs exactly twice in c. A string X is a *closed factor* of w, if X is closed and is a substring of w. We define the *closed factorization* of a string w as follows.

Definition 7 (Closed Factorization)**.** *The closed factorization of string w, denoted $CF(w)$, is a factorization $f_1, \ldots, f_k$ of w such that for any $1 \leq i \leq k$,*

f_i *is the longest prefix of* $w[j..|w|]$ *that is closed, where* $j = |f_1 \cdots f_{i-1}| + 1$.

Example 3. *For string* $w = ababaacbbbcbcc\$$, $CF(w) = ababa, a, cbbbcb, cc, \$$.

We remark that a closed factor f_i is a single character if and only if $|f_1 \cdots f_{i-1}| + 1$ is the rightmost (last) occurrence of character $w[|f_1 \cdots f_{i-1}| + 1]$ in w.

We also define the *longest closed factor array* of string w.

Definition 8 (Longest Closed Factor Array)**.** *The longest closed factor array of w is an array A of integers such that for any $1 \leq i \leq n$, $A[i] = l$ if and only if l is the length of the longest prefix of w that is closed.*

Example 4. *For string* $w = ababaacbbbcbcc\$$, $A = [5, 4, 3, 5, 2, 1, 6, 3, 2, 4, 3, 1, 2, 1, 1]$.

Clearly, given the longest closed factor array A of string w, $CF(w)$ can be computed in $O(n)$ time. However, the algorithm we describe in Section 6.3 to compute A requires $O(n\frac{\log n}{\log\log n})$ time, and so using it to compute $CF(w)$ would also take $O(n\frac{\log n}{\log\log n})$ time overall. In Section 6.2 we present an optimal $O(n)$-time algorithm to compute $CF(w)$ that does not require A.

6.2 Greedy longest closed factorization in linear time

In this section, we present how to compute the closed factorization $CF(w)$ of a given string w. Our high level strategy is to build a data structure that helps us to efficiently compute, for a given position i in w, the longest closed factor starting at i. The core of this data structure is the suffix tree for w, which we decorate in various ways.

Let S be the set of the beginning positions of the longest closed factors in $CF(w)$. For any $i \in S$, let $G = w[i..i + |G| - 1]$ be the longest closed factor of w starting at position i in w.

Let G' be the unique border of the longest closed factor G starting at position i of w, and b_i be its length, i.e., $G' = G[1..b_i] = w[i..i + b_i - 1]$ (if G is a single character, then $G' = \varepsilon$ and $b_i = 0$). The following lemma shows that we can efficiently compute $CF(w)$ if we know b_i for all $i \in S$.

Lemma 19. *Given* b_i *for all* $i \in S$*, we can compute* $CF(w)$ *in a total of* $O(n)$ *time and space independently of the alphabet size.*

Proof. If $b_i = 0$, then $G = w[i]$. Hence, in this case it clearly takes $O(1)$ time and space to compute G.

If $b_i > 1$, then we can compute G in $O(|G|)$ time and $O(b_i)$ space, as follows. We preprocess the border G' of G using the Knuth-Morris-Pratt (KMP) string matching algorithm [49] (see Section 2.3.4). This preprocessing takes $O(b_i)$ time and space. We then search for the first occurrence of G' in $w[i + 1..n]$ (i.e. the next occurrence of the longest border of $G[1, m]$ to the right of the occurrence $w[i..i + b_i - 1]$). The location of the next occurrence tells us where the end of the closed factor is, and so it also tells us $G = w[i..i + |G| - 1]$. The search takes $O(|G|)$ time — i.e. time proportional to the length of the closed factor. Because the sum of the lengths of the closed factors is n, over the whole factorization we take $O(n)$ time and space. The running time and space usage of the algorithm are clearly independent of the alphabet size. □

What remains is to be able to efficiently compute b_i for a given $i \in S$. The following lemma gives an efficient solution to this subproblem:

Lemma 20. *We can preprocess the suffix tree of string w of length n in $O(n)$ time and space, so that b_i for each $i \in S$ can be computed in $O(1)$ time.*

Proof. In each leaf of the suffix tree, we store the beginning position of the suffix corresponding to the leaf. For any internal node v of the suffix tree of w, let $\max(v)$ denote the maximum leaf value in the subtree rooted at v,i.e.,

$$\max(v) = \max\{i \mid w[i..i + \mathit{pathlabel}(v) - 1], 1 \leq i \leq n - \mathit{pathlabel}(v) - 1\}.$$

We can compute $\max(v)$ for every v in a total of $O(n)$ time via a depth first traversal. Next, let P be an array of pointers to suffix tree nodes (to be computed next). Initially every $P[i]$ is set to null. We traverse the suffix tree in pre-order, and for each node v we encounter we set $P[\max(v)] = v$ if $P[\max(v)]$ is null. At the end of the traversal $P[i]$ will contain a pointer to the highest node w in the tree for which i is the maximum leaf value (i.e., i is the rightmost occurrence of $\mathit{pathlabel}(w)$).

We are now able to compute b_i, the length of the unique border of the longest closed factor starting at any given i, as follows. First we retrieve node $v = P[i]$. Observe that, because of the definition of $P[i]$, there are no occurrences of substring $w[i..i + |\mathit{pathlabel}(v)|]$ to the right of i. Let $u = \mathit{parent}(v)$. There are two cases to consider:

- If u is not the root, then observe that there always exists an occurrence of substring $\mathit{pathlabel}(u)$ to the right of position i (otherwise i would be the rightmost occurrence of $\mathit{pathlabel}(u)$, but this cannot be the case since u is higher than v, and we defined $P[i]$ to be the highest node w with $\max(w) = i$). Let j be the the leftmost occurrence of $\mathit{pathlabel}(u)$ to the right of i. Then, the longest closed factor starting at position i is $w[i..j + |\mathit{pathlabel}(u)| - 1]$ (this position j is found by the KMP algorithm as in Lemma 19).

- If u is the root, then it turns out that i is the rightmost occurrence of character $w[i]$ in w. Hence, the longest closed factor starting at position i is $w[i]$.

The thing we have not shown is that $|\mathit{pathlabel}(u)| = b_i$. This is indeed the case, since the set of occurrences of $w[i..j + |\mathit{pathlabel}(u)|]$ (i.e., leaves in the subtree corresponding to the string)

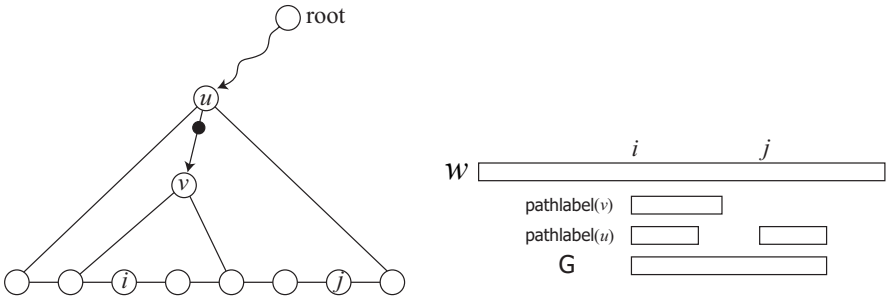

Figure 6.1: Illustration for Lemma 20. We consider the longest closed factor G starting at position i of string w. We retrieve node $P[i] = v$, which implies $\max(v) = i$. Let u be the parent of v. The black circle represents a (possibly implicit) node which represents $w[i..i+pathlabel(u)]$, which has the same set of occurrences as $pathlabel(v)$. Hence $b_i = |pathlabel(u)|$, and therefore $G = w[i..j + pathlabel(u) - 1]$, where j is the leftmost occurrence of $pathlabel(u)$ with $j > i$.

is equivalent to that of $pathlabel(v)$, any substring starting at i that is longer than $|pathlabel(u)|$ does not occur to the right of i and thus b_i cannot be any longer. Hence $|pathlabel(u)| = b_i$. (See also Figure 6.1).

Clearly $v = P[i]$ can be retrieved in $O(1)$ time for a given i, and then $u = parent(v)$ can be obtained in $O(1)$ time from v. This completes the proof. □

The main result of this section follows:

Theorem 13. *Given a string w of length n over an integer alphabet, the closed factorization $CF(w) = f_1, \ldots, f_k$ of w can be computed in $O(n)$ time and space.*

Proof. We compute f_i in ascending order of $i = 1, \ldots, k$. Let s_i be the beginning position of f_i in w, i.e., $s_1 = 1$ and $s_i = |f_1 \cdots f_{i-1}| + 1$ for $1 < i \leq k$. We compute f_1 in $O(|f_1|)$ time and space from b_{s_1} using Lemma 19 and Lemma 20. Assume we have computed the first $i - 1$ factors $f_1, \ldots, f_{i-1}$ for any $1 \leq i < k - 1$. We then compute f_j in $O(|f_j|)$ time and space from b_{f_j}, again using Lemmas 19 and 20. Since $\sum_{i=1}^{k} |f_i| = n$, the proof completes. □

The following is an example of how the algorithm presented in this section computes $CF(w)$ for a given string w.

Example 5. *Consider the running example string $w = ababaacbbbcbcc\$$.*

1. *We begin with node $P[1]$ representing $ababaacbbbcbcc\$$, whose parent represents aba. Hence we get $b_1 = |aba| = 3$. We run the KMP algorithm with pattern aba and find the first factor $f_1 = ababa$.*

2. *We then check node $P[6]$ representing a. Since its parent is the root, we get $b_2 = 0$ and therefore the second factor is $f_2 = a$.*

3. *We then check node $P[7]$ representing $cbbbcbcc\$$, whose parent represents cb. Hence we get $b_3 = |cb| = 2$. We run the KMP algorithm with pattern cb and find the third factor $f_3 = cbbbcb$.*

4. *We then check node $P[13]$ representing $cc\$$, whose parent represents c. Hence we get $b_4 = |c| = 1$. We run the KMP algorithm with pattern c and find the fourth factor $f_4 = cc$.*

5. *We finally check node $P[15]$ representing $\$$. Since its parent is the root, we get $b_5 = 0$ and therefore the fifth factor is $f_5 = \$$.*

Consequently, we obtain $CF(w) = ababa, a, cbbbcb, cc, \$$, which coincides with Example 3.

6.3 Longest closed factor array

A natural extension of the problem in the previous section is to compute the longest closed factor starting at *every* position in w in linear time — not just those involved in the factorization. Formally, we would like to compute the longest closed factor array of w, i.e., an array A of integers such that $A[i] = l$ if and only if l is the length of the longest closed factor starting at position i in w.

Our algorithm for closed factorization computes the longest closed factor starting at a given position in time proportional to the factor's length, and so does not immediately provide a linear time algorithm for computing A; indeed, applying the algorithm naïvely at each position would take $O(n^2)$ time to compute A. In what follows, we present a more efficient solution:

Theorem 14. *Given a string w of length n over an integer alphabet, the closed factor array of w can be computed in $O(n\frac{\log n}{\log\log n})$ time and $O(n)$ space.*

Proof. We extend the data structure of the last section to allow A to be computed in $O(n\frac{\log n}{\log\log n})$ time and $O(n)$ space. The main change is to replace the KMP algorithm scanning in the first algorithm with a data structure that allows us to find the end of the closed factor in time independent of its length.

We first preprocess the suffix array SA for *range successor* queries, building the data structure of Yu, Hon and Wang [77]. A range successor query $rsq_{SA}(s, e, k)$ returns, given a range $[s, e] \subseteq [1, n]$, the smallest value $x \in SA[s..e]$ such that $x > k$, or null if there is no value larger than k in $SA[s..e]$. Yu et al.'s data structure allows range successor queries to be answered in $O(\frac{\log n}{\log \log n})$ time each, takes $O(n)$ space, and $O(n\frac{\log n}{\log \log n})$ time to construct.

Now, to compute the longest closed factor starting at a given position i in w (i.e. to compute $A[i]$) we do the following. First we compute b_i, the length of the border of the longest closed factor starting at i, in $O(1)$ time using Lemma 20.

Recall that in the process of computing b_i we determine the node u having $pathlabel(u) = w[i..i + b_i - 1]$. To determine the end of the closed factor we must find the smallest $j > i$ such that $w[j..j + b_i - 1] = w[i..i + b_i - 1]$. Observe that j, if it exists, is precisely the answer to $rsq_{SA}(u.s, u.e, i)$. (See also the left diagram of Figure 6.1. Assuming that the leaves in the subtree rooted at u are sorted in the lexicographical order, the leftmost and rightmost leaves in the subtree correspond to the $u.s$-th and $u.e$-th entries of SA, respectively. Hence, $j = rsq_{SA}(u.s, u.e, i)$). For each $A[i]$ we spend $O(\frac{\log n}{\log \log n})$ time and so overall the algorithm takes $O(n\frac{\log n}{\log \log n})$ time. The space requirement is clearly $O(n)$. □

We note that recently Navarro and Neckrich [65] described range successor data structures with faster $O(\sqrt{\log n})$-time queries, but straightforward construction takes $O(n \log n)$ time [64], so overall this does not improve the runtime of our algorithm.

Chapter 7

Abelian Regularities

In this chapter, we consider three problems related to Abelian regularities of strings: computing Abelian squares in a given string; computing regular Abelian periods of a given string; and computing longest common Abelian factors of two given strings over. We show that computation of these Abelian regularities can be accelerated via the run-length factorization.

The results in this chapter will be published in [72].

7.1 Notation

For any string $w \in \Sigma^*$, its *Parikh vector* $\mathcal{P}_w$ is an array of length σ such that for any $1 \leq i \leq |\Sigma|$, $\mathcal{P}_w[i]$ is the number of occurrences of each character $c_i \in \Sigma$ in w. For example, for string $w = abaab$ over alphabet $\Sigma = \{a, b\}$, $\mathcal{P}_w = \langle 3, 2 \rangle$. We say that strings x and y are *Abelian equivalent* if $\mathcal{P}_x = \mathcal{P}_y$. Note that $\mathcal{P}_x = \mathcal{P}_y$ iff x and y are permutations of each other. When x is a substring of a permutation of y, then we write $\mathcal{P}_x \subseteq \mathcal{P}_y$. For any Parikh vectors P and Q, let $\mathit{diff}(P, Q) = |\{i \mid P[i] \neq Q[i], 1 \leq i \leq \sigma\}|$.

A string w of length $2k > 0$ is called an *Abelian square* if it is a concatenation of two Abelian equivalent strings of length k each, i.e., $\mathcal{P}_{w[1..k]} = \mathcal{P}_{w[k+1..2k]}$. A string w is said to have a *regular Abelian period* (p, t) if w can be factorized into a sequence $v_1, \ldots, v_s$ of substrings such that $p = |v_1| = \cdots = |v_{s-1}|$, $|v_s| = t$, $\mathcal{P}_{v_i} = \mathcal{P}_{v_1}$ for all $2 \leq i < s$, and $\mathcal{P}_{v_s} \subseteq \mathcal{P}_{v_1}$. For any strings $w_1, w_2 \in \Sigma^*$, if a substring $w_1[i..i + l - 1]$ of w_1 and a substring $w_2[j..j + l - 1]$ of w_2 are Abelian equivalent, then the pair of substrings is said to be a *common Abelian factor* of w_1 and w_2. When the length l is the maximum of such then the pair of substrings is said to be a *longest common Abelian factor* of w_1 and w_2.

The *run length encoding* (*RLE*) of string w of length n, denoted $RLE(w)$, is a compact

representation of w which encodes each maximal character run $w[i..i+p-1]$ by a^p, if (1) $w[j] = a$ for all $i \leq j \leq i+p-1$, (2) $w[i-1] \neq w[i]$ or $i = 1$, and (3) $w[i+p-1] \neq w[i+p]$ or $i+p-1 = n$. E.g., $RLE(aabbbbcccaaa\$) = a^2b^4c^3a^3\$^1$. The *size* of $RLE(w) = a_1^{p_1} \cdots a_m^{p_m}$ is the number m of maximal character runs in w, and each $a_i^{p_i}$ is called an ***RLE factor*** of $RLE(w)$. Notice that $m \leq n$ always holds. Also, since at most m distinct characters can appear in w, in what follows we will assume that $\sigma \leq m$. Even if the underlying alphabet is large, we can sort the characters appearing in w in $O(m \log m)$ time and use this ordering in Parikh vectors. Since all of our algorithms will require at least $O(mn)$ time, this $O(m \log m)$-time preprocessing is negligible.

For any $1 \leq i \leq j \leq n$, let $RLE(w)[i..j] = a_i^{p_i} \cdots a_j^{p_j}$. For convenience let $RLE(w)[i..j] = \varepsilon$ for $i > j$. For $RLE(w) = a_1^{p_1} \cdots a_m^{p_m}$, let $RLE\_Bound(w) = \{1 + \sum_{i=1}^{k} p_k \mid 1 \leq k < m\} \cup \{1, n\}$. For any $1 \leq i \leq n$, let $succ(i) = \min\{j \in RLE\_Bound(w) \mid j > i\}$. Namely, $succ(i)$ is the smallest position in w that is greater than i and is either the beginning position of an RLE factor in w or the last position n in w.

7.2 Computing regular Abelian periods using RLEs

In this section, we propose an algorithm which computes all regular Abelian periods of a given string.

Theorem 15. *Given a string w of length n over an alphabet of size σ, we can compute all regular Abelian periods of w in $O(mn)$ time and $O(n)$ working space, where m is the size of $RLE(w)$.*

Proof. Our algorithm is very simple. We use a single window for each length $d = 1, \ldots, \lfloor \frac{n}{2} \rfloor$. For an arbitrarily fixed d, consider a decomposition $v_1, \ldots, v_s$ of w such that $v_i = w[(i-1)d + 1..id]$ for $1 \leq i \leq \lfloor \frac{n}{d} \rfloor$ and $v_s = w[n - (n \bmod d) + 1..n]$. Each v_i is called a block, and each block of length d is called a complete block.

There are two cases to consider.

Case (a): If w is a unary string (i.e., $RLE(w) = a^n$ for some $a \in \Sigma$). In this case, $(d, (n \bmod d))$ is a regular Abelian period of w for any d. Also, note that this is the only case where $(d, (n \bmod d))$ can be a regular Abelian period of any string of length n with $RLE(v_i) = a^d$ for some complete block v_i. Clearly, it takes a total of $O(n)$ time and $O(1)$ space in this case.

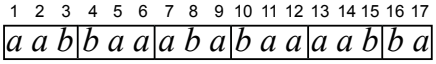

Figure 7.1: $(3, 2)$ is a regular Abelian period of string $w = aabbaaababaaaabbaa$ since $\mathcal{P}_{w[1..3]} = \mathcal{P}_{w[4..6]} = \mathcal{P}_{w[7..9]} = \mathcal{P}_{w[10..12]} = \mathcal{P}_{w[13..15]} \supset \mathcal{P}_{w[16..17]}$.

Case (b): If w contains at least two distinct characters, then observe that a complete block v_i is fully contained in a single RLE factor iff $succ(1 + (i-1)d) = succ(id)$. Let S be an array of length n such that $S[j] = succ(j)$ for each $1 \leq j \leq n$. We precompute this array S in $O(n)$ time by a simple left-to-right scan over w. Using the precomputed array S, we can check in $O(m)$ time if there exists a complete block v_i satisfying $succ(1 + (i-1)d) = succ(id)$; we process each complete block v_i in increasing order of i (from left to right), and stop as soon as we find the first complete block v_i with $succ(1 + (i-1)d) = succ(id)$. If there exists such a complete block, then we can immediately determine that $(d, (n \bmod d))$ is not a regular Abelian period (recall also Case (a) above.)

Assume every complete block v_i overlaps at least two RLE factors. For each v_i, let $m_i \geq 2$ be the number of RLE factors of $RLE(w)$ that v_i overlaps (i.e., m_i is the size of $RLE(v_i)$). We can compute $\mathcal{P}_{v_i}$ in $O(m_i)$ time from $RLE(v_i)$, using the exponents of the elements of $RLE(v_i)$. We can compare $\mathcal{P}_{v_i}$ and $\mathcal{P}_{v_{i-1}}$ in $O(m_i)$ time, since there can be at most m_i distinct characters in v_i and hence it is enough to check the m_i entries of the Parikh vectors. Since there are $\lfloor \frac{n}{d} \rfloor$ complete blocks and each complete block overlaps more than one RLE factor, we have $\lfloor \frac{n}{d} \rfloor \leq \frac{1}{2} \sum_{i=1}^{s-1} m_i$. Moreover, since each RLE factor is counted by a unique m_i or by a unique pair of m_{i-1} and m_i for some i, we have $\sum_{i=1}^{s} m_i \leq 2m$. Overall, it takes $O(\sigma + \frac{n}{d} + \sum_{i=1}^{s} m_i) = O(m)$ time to test if $(d, (n \bmod d))$ is a regular Abelian period of w. Consequently, it takes $O(mn)$ total time to compute all regular Abelian periods of w for all d's in this case. Since we use the array S of length n and we maintain two Parikh vectors of the two adjacent v_{i-1} and v_i for each i, the space requirement is $O(\sigma + n) = O(n)$. □

For example, let $w = aabbaaababaaaabbaa$ and $d = 3$. See also Figure 7.1 for illustration. We have $RLE(w) = a^2b^2a^3b^1a^1b^1a^4b^2a^1$. Then, we compute $\mathcal{P}_{v_1} = \langle 2, 1 \rangle$ from $RLE(v_1) = a^2b^1$, $\mathcal{P}_{v_2} = \langle 2, 1 \rangle$ from $RLE(v_2) = b^1a^2$, $\mathcal{P}_{v_3} = \langle 2, 1 \rangle$ from $RLE(v_3) = a^1b^1a^1$, $\mathcal{P}_{v_4} = \langle 2, 1 \rangle$ from $RLE(v_4) = b^1a^2$, $\mathcal{P}_{v_5} = \langle 2, 1 \rangle$ from $RLE(v_5) = a^2b^1$, and $\mathcal{P}_{v_6} = \langle 1, 1 \rangle$ from $RLE(v_6) = b^1a^1$. Since $\mathcal{P}_{v_i} = \mathcal{P}_{v_1}$ for $1 \leq i \leq 5$ and $\mathcal{P}_{v_6} \subset \mathcal{P}_{v_1}$, $(3, 2)$ is a regular Abelian period of the string w.

7.3 Computing Abelian squares using RLEs

In this section, we describe our algorithm to compute all Abelian squares occurring in a given string w of length n. Our algorithm is based on the algorithm of Cummings and Smyth [24] which computes all Abelian squares in w in $O(n^2)$ time. We will improve the running time to $O(mn)$, where m is the size of $RLE(w)$.

7.3.1 Cummings and Smyth's $O(n^2)$-time algorithm

We recall the $O(n^2)$-time algorithm proposed by Cummings and Smyth [24]. To compute Abelian squares in a given string w, their algorithm aligns two adjacent sliding windows of length d each, for every $1 \leq d \leq \lfloor \frac{n}{2} \rfloor$.

Consider an arbitrary fixed d. For each position $1 \leq i \leq n - 2d + 1$ in w, let L_i and R_i denote the left and right windows aligned at position i. Namely, $L_i = w[i..i + d - 1]$ and $R_i = w[i + d..i + 2d - 1]$. At the beginning, the algorithm computes $\mathcal{P}_{L_1}$ and $\mathcal{P}_{R_1}$ for position 1 in w. It takes $O(d)$ time to compute these Parikh vectors and $O(\sigma)$ time to compute $\mathit{diff}(\mathcal{P}_{L_1}, \mathcal{P}_{R_1})$. Assume $\mathcal{P}_{L_i}$, $\mathcal{P}_{R_i}$, and $\mathit{diff}(\mathcal{P}_{L_i}, \mathcal{P}_{R_i})$ have been computed for position $i \geq 1$, and $\mathcal{P}_{L_{i+1}}$, $\mathcal{P}_{R_{i+1}}$, and $\mathit{diff}(\mathcal{P}_{L_{i+1}}, \mathcal{P}_{R_{i+1}})$ is to be computed for the next position $i + 1$. A key observation is that given $\mathcal{P}_{L_i}$, then $\mathcal{P}_{L_{i+1}}$ for the left window L_{i+1} for the next position $i+1$ can be easily computed in $O(1)$ time, since at most two entries of the Parikh vector can change. The same applies to $\mathcal{P}_{R_i}$ and $\mathcal{P}_{R_{i+1}}$. Also, given $\mathit{diff}(\mathcal{P}_{L_i}, \mathcal{P}_{R_i})$ for the two adjacent windows L_i and R_i for position i, then it takes $O(1)$ time to determine whether or not $\mathit{diff}(\mathcal{P}_{L_{i+1}}, \mathcal{P}_{R_{i+1}}) = 0$ for the two adjacent windows L_{i+1} and R_{i+1} for the next position $i + 1$. Hence, for each d, it takes $O(n)$ time to find all Abelian squares of length $2d$, and thus it takes a total of $O(n^2)$ time for all $1 \leq d \leq \lfloor \frac{n}{2} \rfloor$.

7.3.2 Our $O(mn)$-time algorithm

We propose an algorithm which computes all Abelian squares in a given string w of length n in $O(mn)$ time, where m is the size of $RLE(w)$.

Our algorithm will output consecutive Abelian squares $w[i..i+2d-1]$, $w[i+1..i+2d]$, ..., $w[j..j + 2d - 1]$ of length $2d$ each as a triple $\langle i, j, d \rangle$. A single Abelian square $w[i..i + 2d - 1]$ of length $2d$ will be represented by $\langle i, i, d \rangle$.

For any position i in w, let $beg(L_i)$ and $end(L_i)$ respectively denote the beginning and

ending positions of the left window L_i, and let $beg(R_i)$ and $end(R_i)$ respectively denote the beginning and ending positions of the right window R_i. Namely, $beg(L_i) = i$, $end(L_i) = i + d - 1$, $beg(R_i) = i + d$, and $end(R_i) = i + 2d - 1$. Cummings and Smyth's algorithm described above increases each of $beg(L_i)$, $end(L_i)$, $beg(R_i)$, and $end(R_i)$ one by one, and tests all positions $i = 1, \ldots, n - 2d + 1$ in w. Hence their algorithm takes $O(n)$ time for each window size d.

In what follows, we show that it is indeed enough to check only $O(m)$ positions in w for each window size d. The outline of our algorithm is as follows. As Cummings and Smyth's algorithm, we use two adjacent windows of size d, and slide the windows. However, unlike Cummings and Smyth's algorithm where the windows are shifted by one position, in our algorithm the windows can be shifted by more than one position. The positions that are not skipped and are explicitly examined will be characterized by the RLE of w, and the equivalence of the Parikh vectors of the two adjacent windows for the skipped positions can easily be checked by simple arithmetics.

Now we describe our algorithm in detail. First, we compute $RLE(w)$ and let m be its size. Consider an arbitrarily fixed window length $d \geq 1$.

Initially, we compute $\mathcal{P}_{L_1}$ and $\mathcal{P}_{R_1}$ for position 1. We can compute these Parikh vectors in $O(m)$ time and $O(\sigma)$ space using the same method as in the algorithm of Theorem 15 in Section 7.2.

Then, we describe the steps for positions larger than 1. For each position $i \geq 1$ in a given string w, let $D_1^i = succ(beg(L_i)) - beg(L_i)$, $D_2^i = succ(beg(R_i)) - beg(R_i)$, and $D_3^i = succ(end(R_i) + 1) - end(R_i) - 1$. The *break point* for each position i, denoted $\mathrm{bp}(i)$, is defined by $i + \min\{D_1^i, D_2^i, D_3^i\}$. Assume the left window is aligned at position i in w. Then, we jump to the break point $\mathrm{bp}(i)$ directly from i. In other words, the two windows L_i and R_i are directly shifted to $L_{\mathrm{bp}(i)}$ and $R_{\mathrm{bp}(i)}$, respectively.

It depends on the value of $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i})$ whether there can be an Abelian square between positions i and $\mathrm{bp}(i)$. Note that $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) \neq 1$. Below, we characterize the other cases in detail.

Lemma 21. *Assume* $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) = 0$. *Then, for any* $i < j \leq \mathrm{bp}(i)$, j *is the beginning position of an Abelian square of length* $2d$ *iff* $w[beg(L_i)] = w[beg(R_i)] = w[end(R_i) + 1]$.

Proof. ($\Leftarrow$) By the definition of $\mathrm{bp}(i)$, $w[beg(L_i)] = w[beg(L_j)]$, $w[beg(R_i)] = w[beg(R_j)]$, and $w[end(R_i) + 1] = w[end(R_j) + 1]$ for all $i < j \leq \mathrm{bp}(i)$. Let $c = w[beg(L_i)] = w[beg(R_i)] =$

$w[end(R_i)+1]$. Then we have $w[beg(L_j)] = w[beg(R_j)] = w[end(R_j)+1] = c$. Thus the Parikh vectors of the sliding windows do not change at any position between i and $\mathrm{bp}(i)$. Since we have assumed $\mathcal{P}_{L_i} = \mathcal{P}_{R_i}$, $\mathcal{P}_{L_j} = \mathcal{P}_{R_j}$ for any $i < j \leq \mathrm{bp}(i)$. Thus $w[j..j+2d-1] = L_jR_j$ is an Abelian square of length $2d$ for any $i < j \leq \mathrm{bp}(i)$.

($\Rightarrow$) Since j is the beginning position of an Abelian square of length $2d$, $\mathcal{P}_{L_j} = \mathcal{P}_{R_j}$. Let $c_p = w[beg(L_i)]$, $c_q = w[beg(R_i)]$, and $c_t = w[end(R_i)+1]$. By the definition of $\mathrm{bp}(i)$, $w[beg(L_j)] = c_p$, $w[beg(R_j)] = c_q$, and $w[end(R_j)+1] = c_t$ for any $i < j \leq \mathrm{bp}(i)$. Also, for any $i < j \leq \mathrm{bp}(i)$, $\mathcal{P}_{L_j}[x] = \mathcal{P}_{L_i}[x] - j + i$, $\mathcal{P}_{L_j}[y] = \mathcal{P}_{L_i}[y] + j - i$, $\mathcal{P}_{R_j}[y] = \mathcal{P}_{R_i}[y] - j + i$, and $\mathcal{P}_{R_j}[z] = \mathcal{P}_{R_i}[z] + j - i$. Recall we have assumed that $\mathcal{P}_{L_i} = \mathcal{P}_{R_i}$ and $\mathcal{P}_{L_j} = \mathcal{P}_{R_j}$ for any $i < j \leq \mathrm{bp}(i)$. This is possible only if $c_p = c_q = c_t$, namely, $w[beg(L_j)] = w[beg(R_j)] = w[end(R_j)+1]$.

Lemma 22. *Assume $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) = 2$. Let c_p be the unique character which occurs more in the left window L_i than in the right window R_i, and c_q be the unique character which occurs more in the right window R_i than in the left window L_i. Let $x = \mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] = \mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q] > 0$, and assume $x \leq \min\{D_1^i, D_2^i, D_3^i\}$. Then, $i + x$ is the beginning position of an Abelian square of length $2d$ iff $w[beg(L_i)] = c_p$, $w[beg(R_i)] = c_q = w[end(R_i)+1]$. Also, this is the only Abelian square of length $2d$ beginning at positions between i and $\mathrm{bp}(i)$.*

Proof. ($\Leftarrow$) Since $w[beg(L_i)] = c_p$ and $w[beg(R_i)] = w[end(R_i)+1] = c_q$, we have that $\mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] - z = \mathcal{P}_{L_{i+z}}[p] - \mathcal{P}_{R_{i+z}}[p]$ and $\mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q] + z = \mathcal{P}_{R_{i+z}}[q] = \mathcal{P}_{L_{i+z}}[q]$ for any $1 \leq z \leq \min\{D_1^i, D_2^i, D_3^i\}$. By the definition of x, the Parikh vectors of the sliding windows become equal at position $i + x$.

($\Rightarrow$) Since $x = \mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] = \mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q] > 0$, $\mathcal{P}_{L_{i+x}}[p] = \mathcal{P}_{L_{i+x}}[p]$, and $\mathcal{P}_{L_{i+x}}[q] = \mathcal{P}_{L_{i+x}}[q]$, we have $w[beg(L_i)] = c_p$ and $w[beg(R_i)] = w[end(R_i)+1] = c_q$. From the above arguments, it is clear that $i+x$ is the only position between i and $\mathrm{bp}(i)$ where an Abelian square of length $2d$ can start. □

Lemma 23. *Assume $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) = 2$. Let c_p be the unique character which occurs more in the left window L_i than in the right window R_i, and c_q be the unique character which occurs more in the right window R_i than in the left window L_i. Let $x = \mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] = \mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q] > 0$, and assume $\frac{x}{2} \leq \min\{D_1^i, D_2^i, D_3^i\}$. Then, $i + \frac{x}{2}$ is the beginning position of an Abelian square of length $2d$ iff $w[beg(L_i)] = c_p = w[end(R_i)+1]$, $w[beg(R_i)] = c_q$. Also, this is the only Abelian square of length $2d$ beginning at positions between i and $\mathrm{bp}(i)$.*

Proof. ($\Leftarrow$) Since $w[beg(L_i)] = c_p = w[end(R_i)+1]$ and $w[beg(R_i)] = c_q$, we have that $\mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] - 2z = \mathcal{P}_{L_{i+z}}[p] - \mathcal{P}_{R_{i+z}}[p]$ and $\mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q] + 2z = \mathcal{P}_{R_{i+z}}[q] = \mathcal{P}_{L_{i+z}}[q]$ for any $1 \leq z \leq \min\{D_1^i, D_2^i, D_3^i\}$. Since $\frac{x}{2} \leq \min\{D_1^i, D_2^i, D_3^i\}$, the Parikh vectors of the sliding windows become equal at position $i + \frac{x}{2}$. ($\Rightarrow$) Since $x = \mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] = \mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q] > 0$, $\mathcal{P}_{L_{i+\frac{x}{2}}}[p] = \mathcal{P}_{L_{i+\frac{x}{2}}}[p]$, and $\mathcal{P}_{L_{i+\frac{x}{2}}}[q] = \mathcal{P}_{L_{i+\frac{x}{2}}}[q]$, we have $w[beg(L_i)] = c_p = w[end(R_i)+1]$ and $w[beg(R_i)] = c_q$. From the above arguments, it is clear that $i + \frac{x}{2}$ is the only position between i and $\mathrm{bp}(i)$ where an Abelian square of length $2d$ can start. □

Lemma 24. *Assume* $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) = 3$. *Let* $c_p = w[beg(L_i)]$, $c_{p'} = w[end(R_i)+1]$, *and* $c_q = w[beg(R_i)]$. *Then,* $i+x$ *with* $i < i+x \leq \mathrm{bp}(i)$ *is the beginning position of an Abelian square of length* $2d$ *iff* $0 < x = \mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] = \mathcal{P}_{L_i}[p'] - \mathcal{P}_{R_i}[p'] = \frac{\mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q]}{2} \leq \min\{D_1^i, D_2^i, D_3^i\}$. *Also, this is the only Abelian square of length* $2d$ *beginning at positions between* i *and* $\mathrm{bp}(i)$.

Proof. ($\Leftarrow$) Since $w[beg(L_i)] = c_p$, $w[end(R_i)+1] = c_{p'}$ and $w[beg(R_i)] = c_q$, we have that $\mathcal{P}_{L_i}[p] - z = \mathcal{P}_{L_{i+z}}[p]$, $\mathcal{P}_{L_i}[q] + z = \mathcal{P}_{L_{i+z}}[q]$, $\mathcal{P}_{R_i}[q] - z = \mathcal{P}_{R_{i+z}}[q]$, $\mathcal{P}_{L_i}[q] + z = \mathcal{P}_{L_{i+z}[q]}$ and $\mathcal{P}_{R_i}[p'] + z = \mathcal{P}_{R_{i+z}}[p']$ for any $1 \leq z \leq \min\{D_1^i, D_2^i, D_3^i\}$. Since $x \leq \min\{D_1^i, D_2^i, D_3^i\}$, the Parikh vectors of the sliding windows become equal at position $i + x$ and $i < i + x \leq \mathrm{bp}(i)$.

($\Rightarrow$) Since $i < i + x \leq \mathrm{bp}(i)$, we have $< x \leq \min\{D_1^i, D_2^i, D_3^i\}$. Since $w[beg(L_i)] = c_p$, $w[end(R_i)+1] = c_{p'}$, $w[beg(R_i)] = c_q$, and $\mathcal{P}_{L_{i+x}} = \mathcal{P}_{R_{i+x}}$, we have $x = \mathcal{P}_{L_i}[p] - \mathcal{P}_{R_i}[p] = \mathcal{P}_{L_i}[p'] - \mathcal{P}_{R_i}[p'] = \frac{\mathcal{P}_{R_i}[q] - \mathcal{P}_{L_i}[q]}{2}$.

From the above arguments, it is clear that $i + x$ is the only position between i and $\mathrm{bp}(i)$ where an Abelian square of length $2d$ can start. □

Lemma 25. *Assume* $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) \geq 4$. *Then, there exists no Abelian square of length* $2d$ *beginning at any position* j *with* $i < j \leq \mathrm{bp}(i)$.

Proof. By the definition of $\mathrm{bp}(i)$, we have that $w[beg(L_i)] = w[beg(L_{\mathrm{bp}(i)}) - 1]$, $w[beg(R_i)] = w[beg(R_{\mathrm{bp}(i)}) - 1]$, and $w[end(R_i)] = w[end(R_{\mathrm{bp}(i)}) - 1]$. Since the ending position of the left sliding window is adjacent to the beginning position of the right sliding window, we have $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) - diff(\mathcal{P}_{L_j}, \mathcal{P}_{R_j}) \leq 3$ for any $i \leq j \leq \mathrm{bp}(i)$. Since we have assumed $diff(\mathcal{P}_{L_i}, \mathcal{P}_{R_i}) \geq 4$, we get $diff(\mathcal{P}_{L_j}, \mathcal{P}_{R_j}) \geq 1$. Thus there exist no Abelian squares starting at position j. □

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

Theorem 16. *Given a string w of the length n over an alphabet of size σ, we can compute all Abelian squares in w in $O(mn)$ time and $O(n)$ working space, where m is the size of $RLE(w)$.*

Proof. Consider an arbitrarily fixed window length d. As was explained, it takes $O(m)$ time to compute $\mathcal{P}_{L_1}$, $\mathcal{P}_{R_1}$, and $diff(\mathcal{P}_{L_1}, \mathcal{P}_{R_1})$ for the initial position 1. Suppose that the two windows are aligned at some position $i \geq 1$. Then, our algorithm computes Abelian squares starting at positions between i and $\mathrm{bp}(i)$ using one of Lemma 21, Lemma 22, Lemma 23, Lemma 24, and Lemma 25, depending on the value of $diff(\mathcal{P}_{L_1}, \mathcal{P}_{R_i})$. In each case, all Abelian squares of length $2d$ starting at positions between i and $\mathrm{bp}(i)$ can be computed in $O(1)$ time by simple arithmetics. Then, the left and right windows L_i and R_i are shifted to $L_{\mathrm{bp}(i)}$ and $R_{\mathrm{bp}(i)}$, respectively. Using the array S as in Theorem 15, we can compute $\mathrm{bp}(i)$ in $O(1)$ time for a given position i in w.

Let us analyze the number of times the windows are shifted for each d. Since $\mathrm{bp}(i) = i + \min\{D_1^i, D_2^i, D_3^i\}$, for each position p there can be at most three distinct positions i, j, k such that $p = \mathrm{bp}(i) = \mathrm{bp}(j) = \mathrm{bp}(k)$. Thus, for each d we shift the two adjacent windows at most $3m$ times.

Overall, our algorithm runs in $O(mn)$ time for all window lengths $d = 1, \ldots, \lfloor n/2 \rfloor$. The space requirement is $O(n)$ since we need to maintain the Parikh vectors of the two sliding windows and the array S. □

7.3.3 Example for Computing Abelian squares using RLEs

Here we show some examples on how our algorithm computes all Abelian squares of a given string based on its RLE.

Consider string $w = a^{12}b^4a^3c^2d^2c^2a^2$ over alphabet $\Sigma = \{a, b, c, d\}$ of size 4. Let $d = 4$.

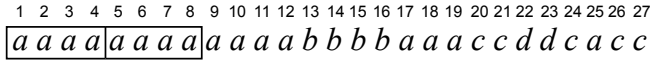

Figure 7.2: $beg(L_1) = 1, beg(R_1) = 5, end(R_1) + 1 = 9, w[beg(L_1)] = w[beg(R_1)] = w[end(R_1) + 1] = a$.

See Figure 7.2 for the initial step of our algorithm, where $i = 1$. As $diff(\mathcal{P}_{L_1}, \mathcal{P}_{R_1}) = 0$, $w[1..8] = aaaaaaaa$ is an Abelian square. Since $\min\{D_1^1, D_2^1, D_3^1\} = \min\{12, 8, 4\} = 4$, the next break point is $\mathrm{bp}(1) = 1 + 4 = 5$. Since $w[beg(L_1)] = w[beg(R_1)] = w[end(R_1) + 1] = a$ and it follows from Lemma 21 that the substrings of length $2d = 8$ between 1 and the break

point are all equal, i.e., $w[1..8] = w[2..9] = w[3..10] = w[4..11] = w[5..12]$, and all of them are Abelian squares. Hence we output a triple $\langle 1, 5, 4\rangle$ representing all these Abelian squares. We update $i \leftarrow \mathrm{bp}(1) = 5$, and proceed to the next step.

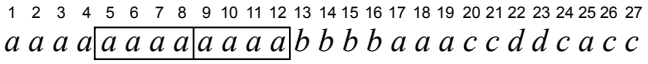

Figure 7.3: $beg(L_5) = 5, beg(R_5) = 9, end(R_5) + 1 = 13, w[beg(L_5)] = w[beg(R_5)] = a, w[end(R_5) + 1] = b$.

Next, see Figure 7.3 where the left window has been shifted to $L_5 = w[5..6] = aaaa$ and the right window has been shifted to $R_5 = w[8..12] = aaaa$. Since $\min\{D_1^5, D_2^5, D_3^5\} = \min\{8, 4, 4\} = 4$, the next break point is $\mathrm{bp}(5) = 5 + 4 = 9$. Since $\mathcal{P}_{L_5} = \mathcal{P}_{R_5}$ and $w[beg(L_5)] = w[beg(R_5)] = a \neq w[end(R_5) + 1] = b$, it follows from Lemma 21 that there are no Abelian squares between 5 and the break point 9. We update $i \leftarrow \mathrm{bp}(5) = 9$, and proceed to the next step.

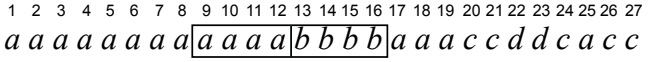

Figure 7.4: $beg(L_9) = 9, beg(R_9) = 13, end(R_9) + 1 = 17, w[beg(L_9)] = a, w[beg(R_9)] = b, w[end(R_9) + 1] = a$.

Next, see Figure 7.4 where the left window has been shifted to $L_9 = w[9..12] = aaaa$ and the right window has been shifted to $R_9 = w[13..16] = bbbb$. Since $\min\{D_1^9, D_2^9, D_3^9\} = \min\{4, 4, 3\} = 3$, the next break point is $\mathrm{bp}(9) = 9 + 3 = 12$. Since $diff(\mathcal{P}_{L_9}, \mathcal{P}_{R_9}) = 2$, $w[beg(L_9)] = w[end(R_9) + 1] = a \neq w[beg(R_9)] = b$, and $\frac{\mathcal{P}_{L_9}[a] - \mathcal{P}_{R_9}[a] = \mathcal{P}_{R_9}[b] - \mathcal{P}_{L_9}[b]}{2} = 2 \leq \min\{D_1^9, D_2^9, D_3^9\} = 3$, it follows from Lemma 23 that $w[11..18]$ is the only Abelian square of length $2d = 8$ starting at positions between 9 and 12. We hence output $\langle 11, 11, 4\rangle$. We update $i \leftarrow \mathrm{bp}(9) = 12$, and proceed to the next step.

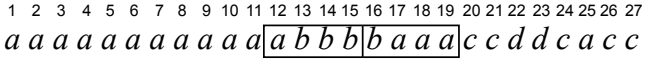

Figure 7.5: $beg(L_{12}) = 12, beg(R_{12}) = 16, end(R_{12}) + 1 = 20, w[beg(L_{12})] = a, w[beg(R_{12})] = b, w[end(R_{12}) + 1] = c$.

Next, see Figure 7.5 where the left window has been shifted to $L_{12} = w[12..15] = abbb$ and the right window has been shifted to $R_{12} = w[16..19] = baaa$. Since $\min\{D_1^{12}, D_2^{12}, D_3^{12}\} =$

$\min\{1, 1, 1\} = 1$, the next break point is $\mathrm{bp}(12) = 12 + 1 = 13$. Since $\mathit{diff}(\mathcal{P}_{L_{12}}, \mathcal{P}_{R_{12}}) = 3$ and $w[\mathit{beg}(L_{12})] = a \neq w[\mathit{beg}(R_{12})] = b \neq w[\mathit{end}(R_{12}) + 1] = c$, it follows from Lemma 22 and Lemma 23 that there are no Abelian squares starting at positions between 12 and 13. We update $i \leftarrow \mathrm{bp}(12) = 13$, and proceed to the next step.

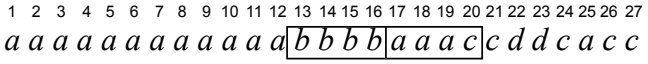

Figure 7.6: $\mathit{beg}(L_{13}) = 13$, $\mathit{beg}(R_{13}) = 17$, $\mathit{end}(R_{13})+1 = 21$, $w[\mathit{beg}(L_{13})] = b$, $w[\mathit{beg}(R_{13})] = a$, $w[\mathit{end}(R_{13}) + 1] = c$.

Next, see Figure 7.6 where the left window has been shifted to $L_{13} = w[13..16] = bbbb$ and the right window has been shifted to $R_{13} = w[17..20] = aaac$. Since $\min\{D_1^{13}, D_2^{13}, D_3^{13}\} = \min\{4, 3, 1\} = 1$, the next break point is $\mathrm{bp}(13) = 13 + 1 = 14$. Since $\mathit{diff}(\mathcal{P}_{L_{13}}, \mathcal{P}_{R_{13}}) = 3$ and $\mathcal{P}_{L_{13}}[b] - \mathcal{P}_{R_{13}}[b] = 4 \neq -1 = \mathcal{P}_{L_{13}}[c] - \mathcal{P}_{R_{13}}[c]$, it follows from Lemma 24 that 14 is not the beginning position of an Abelian square of length $2d = 8$. We update $i \leftarrow \mathrm{bp}(13) = 14$, and proceed to the next step.

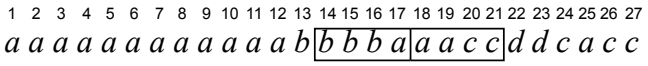

Figure 7.7: $\mathit{beg}(L_{14}) = 14$, $\mathit{beg}(R_{14}) = 18$, $\mathit{end}(R_{14})+1 = 22$, $w[\mathit{beg}(L_{14})] = b$, $w[\mathit{beg}(R_{14})] = a$, $w[\mathit{end}(R_{14}) + 1] = d$.

Next, see Figure 7.7 where the left window has been shifted to $L_{14} = w[14..17] = bbba$ and the right window has been shifted to $R_{14} = w[18..21] = aacc$. Since $\min\{D_1^{14}, D_2^{14}, D_3^{14}\} = \min\{3, 2, 2\} = 2$, the next break point is $\mathrm{bp}(14) = 14 + 2 = 16$. Since $\mathit{diff}(\mathcal{P}_{L_{14}}, \mathcal{P}_{R_{14}}) = 3$ and $\mathcal{P}_{L_{14}}[b] - \mathcal{P}_{R_{14}}[b] = 3 \neq -1 = \mathcal{P}_{L_{14}}[c] - \mathcal{P}_{R_{14}}[c]$, it follows from Lemma 24 that there are no Abelian squares starting at positions between 14 and 16. We update $i \leftarrow \mathrm{bp}(14) = 16$, and proceed to the next step.

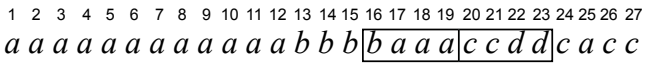

Figure 7.8: $\mathit{beg}(L_{16}) = 16$, $\mathit{beg}(R_{16}) = 20$, $\mathit{end}(R_{16})+1 = 24$, $w[\mathit{beg}(L_{16})] = b$, $w[\mathit{beg}(R_{16})] = w[\mathit{end}(R_{16}) + 1] = c$

Next, see Figure 7.8 where the left window has been shifted to $L_{16} = w[16..19] = baaa$ and the right window has been shifted to $R_{16} = w[20..23] = ccdd$. Since $\min\{D_1^{16}, D_2^{16}, D_2^{16}\} =$

$\min\{1, 2, 2\} = 1$, the next break point is $\mathrm{bp}(16) = 16 + 1 = 17$. Since $\mathit{diff}(\mathcal{P}_{L_{16}}, \mathcal{P}_{R_{16}}) = 3$ and $\mathcal{P}_{L_{16}}[b] - \mathcal{P}_{R_{16}}[b] = 1 \neq -2 = \mathcal{P}_{L_{16}}[c] - \mathcal{P}_{R_{16}}[c]$, it follows from Lemma 24 that 16 is not the beginning position of an Abelian square of length $2d = 8$. We update $i \leftarrow \mathrm{bp}(16) = 17$, and proceed to the next step.

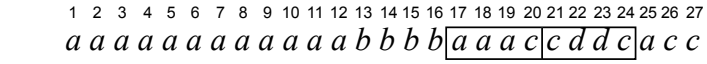

Figure 7.9: $\mathit{beg}(L_{17}) = 17, \mathit{beg}(R_{17}) = 21, \mathit{end}(R_{17}) + 1 = 25, w[\mathit{beg}(L_{17})] = a, w[\mathit{beg}(R_{17})] = c, w[\mathit{end}(R_{17}) + 1] = a$

Next, see Figure 7.9 where the left window has been shifted to $L_{17} = w[17..20] = aaac$ and the right window has been shifted to $R_{17} = w[21..24] = cddc$. Since $\min\{D_1^{17}, D_2^{17}, D_2^{17}\} = \min\{3, 1, 1\} = 1$, the next break point is $\mathrm{bp}(17) = 17 + 1 = 18$. Since $\mathit{diff}(\mathcal{P}_{L_{17}}, \mathcal{P}_{R_{17}}) = 3$ and $\mathcal{P}_{L_{17}}[a] - \mathcal{P}_{R_{17}}[a] = 3 \neq -2 = \mathcal{P}_{L_{17}}[c] - \mathcal{P}_{R_{17}}[c]$, it follows from Lemma 24 that 17 is not the beginning position of an Abelian square of length $2d = 8$. We update $i \leftarrow \mathrm{bp}(17) = 18$, and proceed to the next step.

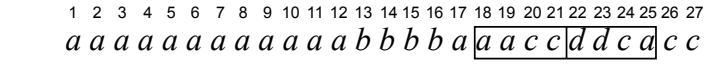

Figure 7.10: $\mathit{beg}(L_{18}) = 18, \mathit{beg}(R_{18}) = 22, \mathit{end}(R_{18}) + 1 = 26, w[\mathit{beg}(L_{18})] = a, w[\mathit{beg}(R_{18})] = d, w[\mathit{end}(R_{18}) + 1] = c$

Next, see Figure 7.10 where the left window has been shifted to $L_{18} = w[18..21] = aacc$ and the right window has been shifted to $R_{18} = w[20..25] = ddcc$. Since $\min\{D_1^{18}, D_2^{18}, D_2^{18}\} = \min\{2, 2, 2\} = 2$, the next break point is $\mathrm{bp}(18) = 18 + 2 = 20$. Since $\mathit{diff}(\mathcal{P}_{L_{18}}, \mathcal{P}_{R_{18}}) = 3$, we use Lemma 24. Since $\mathcal{P}_{L_{18}}[a] - \mathcal{P}_{R_{18}}[a] = \mathcal{P}_{L_{18}}[c] - \mathcal{P}_{R_{18}}[c] = \frac{\mathcal{P}_{R_{18}}[d] - \mathcal{P}_{L_{18}}[d]}{2} = 1 \leq \min\{D_1^{18}, D_2^{18}, D_3^{18}\} = 2$, it follows from Lemma 24 that $w[19..26]$ is an Abelian square of length $2d = 8$. We hence output $\langle 19, 19, 4\rangle$. We update $i \leftarrow \mathrm{bp}(19) = 20$, and proceed to the next step.

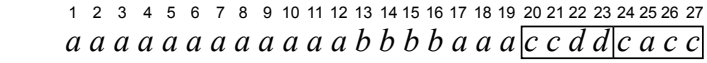

Figure 7.11: $\mathit{beg}(L_{20}) = 20, \mathit{beg}(R_{20}) = 24, w[\mathit{beg}(L_{20})] = c, w[\mathit{beg}(R_{20})] = c$

Next, see Figure 7.11 where the left window has been shifted to $L_{20} = w[20..23] = ccdd$ and the right window has been shifted to $R_{20} = w[24..27] = cacc$. Since $\mathit{diff}(\mathcal{P}_{L_{20}}, \mathcal{P}_{R_{20}}) = 3$

the right end of the right window has reached the last positions of the input string, the algorithm terminates here. Recall that this algorithm computed all the Abelian squares of length $2d = 8$ in this string.

7.4 Computing longest common Abelian factors using RLEs

In this section, we introduce our RLE-based algorithm which computes longest common Abelian factors of two given strings w_1 and w_2. Formally, we solve the following problem. Let $n = \min\{|w_1|, |w_2|\}$. Given two strings w_1 and w_2, compute the length $l = \max\{d \in [1, n] \mid 1 \leq \exists i \leq |w_1|, 1 \leq \exists k \leq |w_2| \text{ s.t. } \mathcal{P}_{w_1[i..i+d-1]} = \mathcal{P}_{w_2[k..k+d-1]}\}$ of the longest common Abelian factor(s) of w_1 and w_2, together with a pair (i, k) of positions on w_1 and w_2 such that $\mathcal{P}_{w_1[i..i+l-1]} = \mathcal{P}_{w_2[k..k+l-1]}$.

Our algorithm uses an idea from Alattabi et al.'s algorithm [2]. For each window size d, their algorithm computes the Parikh vectors of all substrings of w_1 and w_2 of length d in $O(\sigma n)$ time, using two windows of length d each. Then they sort the Parikh vectors in $O(\sigma n)$ time, and output the largest d for which common Parikh vectors exist for w_1 and w_2, together with the lists of respective occurrences of longest common Abelian factors. The total time requirement is clearly $O(\sigma n^2)$.

Our algorithm is different from Alattabi et al.'s algorithm in that (1) we use RLEs of strings w_1 and w_2 and (2) we avoid to sort the Parikh vectors. As in the previous sections, for a given window length d $(1 \leq n)$, we shift two windows of length d over both $RLE(w_1)$ and $RLE(w_2)$, and stops when we reach a break point of $RLE(w_1)$ or $RLE(w_2)$. We then check if there is a common Abelian factor in the ranges of w_1 and w_2 we are looking at.

Since we use a single window for each of the input strings w_1 and w_2, we need to modify the definition of the break points. Let U_i and V_k be the sliding windows for w_1 and w_2 that are aligned at position i of w_1 and at position k of w_2, respectively. For each position $i \geq 1$ in w_1, let $\mathrm{bp}_1(i) = i + \min\{D_1^i, D_2^i\}$, where $D_1^i = succ(beg(U_i)) - i$ and $D_2^i = succ(end(U_i)) - i$. For each position $k \geq 1$ in w_2, $\mathrm{bp}_2(k)$ is defined analogously. Let $p_l = beg(U_i)$, $p_r = end(U_i) + 1$, $q_l = beg(V_k)$ and $q_r = end(V_k) + 1$.

Consider an arbitrarily fixed window length d. Assume that we have just shifted the window on w_1 from position i (i.e., $U_i = w_1[i..i + d - 1]$) to the break point $\mathrm{bp}_1(i)$ (i.e., $U_{\mathrm{bp}_1(i)} = w_1[\mathrm{bp}_1(i)..\mathrm{bp}_1(i) + d - 1]$). Let $c_{p_l} = w_1[i]$ and $c_{p_r} = w_1[i + d]$ (see also Figure 7.12).

For characters c_{p_l} and c_{p_r}, we consider the minimum and maximum numbers of their oc-

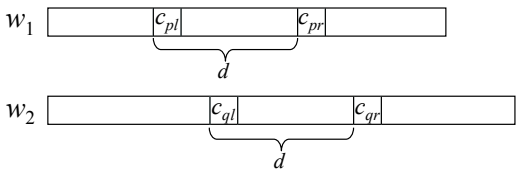

Figure 7.12: Conceptual drawing of c_{p_l}, c_{p_r}, c_{q_r}, and c_{q_l}.

currences during the slide from position i to $\mathrm{bp}_1(i)$. Let $min(p_l) = \mathcal{P}_{w_1[\mathrm{bp}_1(i)..\mathrm{bp}_1(i)+d-1]}[p_l]$, $max(p_l) = \mathcal{P}_{w_1[i..i+d-1]}[p_l]$, $min(p_r) = \mathcal{P}_{w_1[i..i+d-1]}[p_r]$ and $max(p_r) = \mathcal{P}_{w_1[\mathrm{bp}_1(i)..\mathrm{bp}_1(i)+d-1]}[p_r]$. We will use these values to determine if there is a common Abelian factor of length d for w_1 and w_2.

Also, assume that we have just shifted the window on w_2 from position k (i.e., $V_k = w_2[k..k+d-1]$) to the break point $\mathrm{bp}_2(k)$ (i.e., $V_{\mathrm{bp}_2(k)} = w_2[\mathrm{bp}_2(k)..\mathrm{bp}_2(k)+d-1]$). Let $c_{q_l} = w_2[k]$ and $c_{q_r} = w_2[k+d]$ (see also Figure 7.12). For characters c_{q_l} and c_{q_r}, we also consider the minimum and maximum numbers of occurrences of of these characters during the slide from position k to $\mathrm{bp}_2(k)$. Let $min(q_l) = \mathcal{P}_{w_2[\mathrm{bp}_2(k)..\mathrm{bp}_2(k)+d-1]}[q_l]$, $max(q_l) = \mathcal{P}_{w_2[k..k+d-1]}[q_l]$, $min(q_r) = \mathcal{P}_{w_2[k..k+d-1]}[q_r]$ and $max(q_r) = \mathcal{P}_{w_2[\mathrm{bp}_2(k)..\mathrm{bp}_2(k)+d-1]}[q_r]$.

Let m be the total size of $RLE(w_1)$ and $RLE(w_2)$, and l be the length of longest common Abelian factors of w_1 and w_2. Our algorithm computes an $O(m^2)$-size representation of every pair (i,k) of positions for which $(w_1[i..i+l-1], w_2[k..k+l-1])$ is a longest common Abelian factor of w_1 and w_2.

In the lemmas which follow, we assume that $\mathcal{P}_{w_1[i..i+d-1]}[v] = \mathcal{P}_{w_2[k..k+d-1]}[v]$ for any $v \in \{1,..,\sigma\} \setminus \{p_l, p_r, q_l, q_r\}$. This is because, if this condition is not satisfied, then there cannot be an Abelian common factor of length d for positions between i to $\mathrm{bp}_1(i)$ in w_1 and position between k to $\mathrm{bp}_2(k)$ in w_2.

Lemma 26. *Assume $c_{p_l} = c_{p_r}$ and $c_{q_l} = c_{q_r}$. Then, for any pair of positions $i \le i' \le \mathrm{bp}_1(i)$ and $k \le k' \le \mathrm{bp}_2(k)$, $(w_1[i'..i'+d-1], w_2[k'..k'+d-1])$ is an Abelian common factor of length d iff $\mathcal{P}_{w_1[i..i+d-1]} = \mathcal{P}_{w_2[k..k+d-1]}$.*

Proof. Since $c_{p_l} = c_{p_r}$ and $c_{q_l} = c_{q_r}$, the Parikh vectors of the sliding windows do not change during the slides from i to $\mathrm{bp}_1(i)$ and from k to $\mathrm{bp}_2(k)$. Thus the lemma holds. □

Lemma 27. *Assume $c_{p_l} = c_{q_l} \neq c_{p_r} = c_{q_r}$. There is a common Abelian common factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ of length d iff $0 \le x \le \mathrm{bp}_1(i) - i$, $0 \le y \le \mathrm{bp}_2(k) - k$ and $x - y = max(p_l) - max(q_l) = min(q_r) - min(p_r)$.*

Proof. During the slide of the window on w_1, the number of occurrences of c_{p_l} decreases and that of c_{p_r} increases. That is, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - x = max(p_l) - x$ and $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = \mathcal{P}_{w_1[i..i+d-1]}[p_r] + x = min(p_r) + x$. On the other hand, during the slide of the window on w_2, the number of occurrence of c_{q_l} decreases and that of c_{q_r} increases. That is, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y = max(q_l) - y$ and $\mathcal{P}_{w_2[k+y..k+y+d-1]}[p_r] = \mathcal{P}_{w_2[k..k+d-1]}[q_r] + y = min(q_r) + y$.

Assume a pair $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ is a common Abelian factor of length d. Then, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l]$ and $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r]$, that is, $max(p_l) - x = max(q_l)$
$- y$ and $min(p_r) + x = min(q_r) + y$. Therefore $x - y = max(p_l) - max(q_l) = min(q_r) - min(p_r)$.

Assume that $x - y = max(p_l) - max(q_l) = min(q_r) - min(p_r)$. Then, we have that $max(p_l) - max(q_l) = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - \mathcal{P}_{w_2[k..k+d-1]}[q_l] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] + x - \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] - y = x - y$, that is, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l]$. Also, we have that $min(q_r) - min(p_r) = \mathcal{P}_{w_2[k..k+d-1]}[q_r] - \mathcal{P}_{w_1[i..i+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] - y - \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] + x = x - y$, that is, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r]$. Therefore, a pair $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ is a common Abelian factor of w_1 and w_2. □

Lemma 28. *Assume $c_{p_r} \neq c_{p_l} = c_{q_l} \neq c_{q_r}$ and $c_{p_r} \neq c_{q_r}$. There is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ of length diff $\mathrm{bp}_1(i)-i \geq x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r) \geq 0$, $\mathrm{bp}_2(k) - k \geq y = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r) \geq 0$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_l] - x = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y$.*

Proof. During the slides of the windows on w_1 and w_2, the numbers of occurrences of c_{q_r} in w_1 and c_{p_r} in w_2 do not change.

Assume there is a common Abelian factor $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ of length d. Clearly $0 \leq x \leq \mathrm{bp}_1(i) - i$ and $0 \leq y \leq \mathrm{bp}_2(k) - k$. Then, we have $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[p_r]$, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_r]$ and $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l]$, that is, $min(p_r)+x = \mathcal{P}_{w_2[k..k+d-1]}[p_r]$, $min(q_r)+y = \mathcal{P}_{w_1[i..i+d-1]}[q_r]$ and $\mathcal{P}_{w_1[i..i+d-1]}\ [p_l] - x = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y$. Consequently, we obtain $x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r)$ and $y = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r)$.

Assume that $\mathrm{bp}_1(i) - i \geq x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r) \geq 0$, $\mathrm{bp}_2(k) - k \geq y = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r) \geq 0$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_l] - x = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y$. Then, we

have that $min(p_r) + x = \mathcal{P}_{w_1[i..i+d-1]}[p_r] + x = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = \mathcal{P}_{w_2[k..k+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[p_r]$, $min(q_r) + y = \mathcal{P}_{w_2[k..k+d-1]}[q_r] + y = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_r]$ and $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l]$. Therefore, a pair $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ is a common Abelian factor of w_1 and w_2. □

Lemma 29. *Assume $c_{p_l} \neq c_{p_r} = c_{q_r} \neq c_{q_l}$ and $c_{p_l} \neq c_{q_l}$. There is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ of length d iff $x = max(p_l) - \mathcal{P}_{w_2[k..k+d-1]}[p_l] \geq 0$, $y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] \geq 0$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_r] + x = \mathcal{P}_{w_2[k..k+d-1]}[q_r] + y$.*

Lemma 29 can be proved by a similar argument to the proof of Lemma 28.

Lemma 30. *Assume $c_{p_l} = c_{q_r} \neq c_{p_r} = c_{q_l}$. There is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ of length d iff $x+y = min(p_r) - max(q_l) = max(q_l) - min(p_r)$, $0 \leq x \leq \mathrm{bp}_1(i) - i$ and $0 \leq y \leq \mathrm{bp}_2(k) - k$.*

Proof. When the window on w_1 slides by x positions, the occurrence of c_{p_l} in the window decreases by x and the occurrence of c_{p_r} in the window increases by x. When the window on w_2 slides by y positions, the occurrence of c_{q_l} in the window decreases by y and the occurrence of c_{q_r} in the window increases by y.

Assume there is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$. Then $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = \mathcal{P}_{w_1[i..i+d-1]}[p_r] + x = min(p_r) + x$, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y = max(q_l) - y$, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - x = max(p_l) - x$ and $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = \mathcal{P}_{w_2[k..k+d-1]}[q_r] + y = min(q_r) + y$. Therefore $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] \Leftrightarrow x + y = max(q_l) - min(p_r)$ and $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] \Leftrightarrow x + y = max(p_l) - min(q_r)$.

Assume $x+y = max(q_l) - min(p_r) = max(p_l) - min(q_r)$. Clearly $0 \leq x \leq \mathrm{bp}_1(i) - i$ and $0 \leq y \leq \mathrm{bp}_2(k) - k$. Then $max(q_l) - y = min(p_r) + x$ and $max(p_l) - x = min(q_r) + y$, that is, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r]$ and $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r]$. Therefore a pair $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ is a common Abelian factor of w_1 and w_2. □

Lemma 31. *Assume c_{p_l}, c_{p_r}, c_{q_l} and c_{q_r} are mutually distinct. There is a common Abelian factor $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ of length d iff $0 \leq x = max(p_l) - \mathcal{P}_{w_2[k..k+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r) \leq \mathrm{bp}_1(i) - i$ and $0 \leq y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r) \leq \mathrm{bp}_2(k) - k$.*

Proof. During the slides, the numbers of occurrences of c_{q_l} and c_{q_r} in the window on w_1 do not change, and those of c_{p_l} and c_{p_r} in the window on w_2 do not change.

Assume there is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$. Then, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = min(p_r) + x = \mathcal{P}_{w_2}[p_r]$, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = max(p_l) - x = \mathcal{P}_{w_2}[p_l]$, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = min(q_r) + y = \mathcal{P}_{w_1}[q_r]$ and $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = max(q_l) - y = \mathcal{P}_{w_1}[q_l] \Leftrightarrow 0 \leq x = max(p_l) - \mathcal{P}_{w_2}[p_l] = \mathcal{P}_{w_2}[p_r] - min(p_r) \leq \mathrm{bp}_1(i) - i$ and $0 \leq y = max(q_l) - \mathcal{P}_{w_1}[q_l] = \mathcal{P}_{w_1}[q_r] - min(q_r) \leq \mathrm{bp}_2(k) - k$.

Assume $0 \leq x = max(p_l) - \mathcal{P}_{w_2[k..k+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r) \leq \mathrm{bp}_1(i) - i$ and $0 \leq y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r) \leq \mathrm{bp}_2(k) - k$. Then, $x = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - \mathcal{P}_{w_2[k..k+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - \mathcal{P}_{w_2[k..k+d-1]}[q_r]$ and $y = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - \mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - \mathcal{P}_{w_2[k..k+d-1]}[q_r]$. That is, $\mathcal{P}_{w_2[k..k+d-1]}[p_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[p_l] = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - x = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l]$, $\mathcal{P}_{w_2[k..k+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[p_r] = \mathcal{P}_{w_1[i..i+d-1]}[p_r] + x = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r]$, $\mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_l] = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l]$, and $\mathcal{P}_{w_1[i..i+d-1]}[q_r] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_r] = \mathcal{P}_{w_2[k..k+d-1]}[q_r] + y = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r]$. Therefore, a pair $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ is a common Abelian factor of w_1 and w_2. □

Lemma 32. *Assume $c_{q_l} \neq c_{p_l} = c_{p_r} \neq c_{q_r}$ and $c_{q_l} \neq c_{q_r}$. There is a common Abelian factor $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ of length d iff $0 \leq x \leq \mathrm{bp}_1(i) - i$, $0 \leq y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r) \leq \mathrm{bp}_2(k) - k$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_l]$.*

Proof. During the slide, the number of occurrences of c_{p_l} $(= c_{p_r})$ in the window on w_1 does not change.

Assume that there is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$. Clearly $0 \leq x \leq \mathrm{bp}_1(i) - i$ and $0 \leq y \leq \mathrm{bp}_2(k) - k$. Then, it holds that $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = max(q_l) - y = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_l]$, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = min(q_r) + y = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_r]$ and $\mathcal{P}_{w_2[k+y..k+y+d-1]}[p_l] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l]$, that is, $0 \leq y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r)$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_l]$.

Assume $y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r)$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_l]$. Then, $\mathcal{P}_{w_2[k..k+d-1]}[q_l] - y = \mathcal{P}_{w_1[i..i+d-1]}[q_l]$ and $\mathcal{P}_{w_2[k..k+d-1]}[q_r] + y = \mathcal{P}_{w_1[i..i+d-1]}[q_r]$, that is, $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = \mathcal{P}_{w_1[i..i+d-1]}[q_l]$ and $\mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = \mathcal{P}_{w_1[i..i+d-1]}[q_r]$. Therefore, a pair $(w_1[i + x..i + x + d - 1], w_2[k + y..k + y + d - 1])$ is a common Abelian factor of length d of w_1 and w_2. □

Lemma 33. *Assume* $c_{p_l} \neq c_{q_l} = c_{q_r} \neq c_{p_r}$ *and* $c_{p_l} \neq c_{p_r}$. *There is a common Abelian factor* $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ *of length* d *iff* $0 \leq y \leq \mathrm{bp}_2(k) - k$ *and* $x = max(p_l) - \mathcal{P}_{w_2[k..k+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r) \geq 0$.

Lemma 33 can be proved by a similar argument to the proof of Lemma 32.

Lemma 34. *Assume* $c_{p_r} \neq c_{p_l} = c_{q_r} \neq c_{q_l}$ *and* $c_{p_r} \neq c_{q_l}$. *There is a common Abelian factor* $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ *of length* d *iff* $0 \leq x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r) \leq \mathrm{bp}_1(i) - i$, $0 \leq y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l] \leq \mathrm{bp}_2(k) - k$ *and* $x + y = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - \mathcal{P}_{w_2[k..k+d-1]}[q_r] = max(p_l) - min(q_r)$.

Proof. During the slides of the windows, the number of occurrences of c_{q_l} in the window on w_1 and that of c_{p_r} in the window on w_2 do not change.

Assume there is a common Abelian factor $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$. Then, $\mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l] = max(q_l) - y$, $\mathcal{P}_{w_2[k..k+d-1]}[p_r] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r] = min(p_r)+x$, $\mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = min(p_l)+x = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r] = max(q_r) - y$, that is, $y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l]$, $x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r)$ and $x + y = max(p_l) - min(q_r)$.

Assume $y = max(q_l) - \mathcal{P}_{w_1[i..i+d-1]}[q_l]$, $x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - min(p_r)$ and $x + y = max(p_l) - min(q_r)$. Then, $y = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - \mathcal{P}_{w_1[i..i+d-1]}[q_l]$, $x = \mathcal{P}_{w_2[k..k+d-1]}[p_r] - \mathcal{P}_{w_1[i..i+d-1]}[p_r]$ and $x + y = \mathcal{P}_{w_1[i..i+d-1]}[p_l] - \mathcal{P}_{w_2[k..k+d-1]}[q_r]$, that is, $\mathcal{P}_{w_1[i..i+d-1]}[q_l] = \mathcal{P}_{w_1[i+x..i+x+d-1]}[q_l] = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - y = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_l]$, $\mathcal{P}_{w_2[k..k+d-1]}[p_r] = \mathcal{P}_{w_2[k+y..k+y+d-1]}[p_r] = \mathcal{P}_{w_1[i..i+d-1]}\ [p_r] + x = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_r]$ and $\mathcal{P}_{w_1[i..i+d-1]}[p_l] - x = \mathcal{P}_{w_1[i+x..i+x+d-1]}[p_l] = \mathcal{P}_{w_2[k..k+d-1]}[q_r] + y = \mathcal{P}_{w_2[k+y..k+y+d-1]}[q_r]$. Therefore, a pair $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ is a common Abelian factor of length d of w_1 and w_2. □

Lemma 35. *Assume* $c_{p_l} \neq c_{q_l} = c_{p_r} \neq c_{q_r}$ *and* $c_{p_l} \neq c_{q_r}$. *There is a common Abelian factor* $(w_1[i+x..i+x+d-1], w_2[k+y..k+y+d-1])$ *of length* d *iff* $0 \leq x = max(p_l) - \mathcal{P}_{w_2[k..k+d-1]}[p_l] \leq \mathrm{bp}_1(i) - i$, $0 \leq y = \mathcal{P}_{w_1[i..i+d-1]}[q_r] - min(q_r) \leq \mathrm{bp}_2(k) - k$ *and* $x + y = \mathcal{P}_{w_2[k..k+d-1]}[q_l] - \mathcal{P}_{w_1[i..i+d-1]}[p_r] = max(q_l) - min(p_r)$.

Lemma 35 can be proved by a similar argument to the proof of Lemma 34.

Theorem 17. *Given two strings* w_1 *and* w_2, *we can compute an* $O(m^2)$*-size representation of all longest common Abelian factors of* w_1 *and* w_2 *in* $O(m^2 n)$ *time with* $O(\sigma)$ *working space, where* m *and* n *are the total size of the RLEs and the total length of* w_1 *and* w_2, *respectively.*

Proof. The correctness follows from Lemmas 26–35.

Let m_1, m_2 be the sizes of $RLE(w_1)$ and $RLE(w_2)$, respectively. Let $n_{\min} = \min\{|w_1|, |w_2|\}$. For each fixed window size d, the window for w_1 shifts over w_1 $O(m_1)$ times. For each shift of the window for w_1, the window for w_2 shifts over w_2 $O(m_2)$ times. Thus, we have $O(m_1 \cdot m_2 \cdot n_{\min})$ total shifts. Since all the conditions in Lemmas 26–35 can be tested in $O(1)$ time each by simple arithmetic, the total time complexity is $O(m_1 m_2 n_{\min} + n)$, where the n term denotes the cost to compute $RLE(w_1)$ and $RLE(w_2)$. Thus, it is clearly bounded by $O(m^2 n)$. Next, we focus on the output size. Let l be the length of the longest common Abelian factors of w_1 and w_2. Using Lemmas 27–35, for each pair of the shifts of the two windows we can compute an $O(1)$-size representation of the longest common Abelian factors found. Since there are $O(m_1 \cdot m_2)$ shifts for window length l, the output size is bounded by $O(m^2)$. The working space is $O(\sigma)$, since we only need to maintain two Parikh vectors for the two sliding windows. □

Examples.

We show an example of how our algorithm computes a common Abelian factor of length 4 for two input strings $w_1 = aaaaacbbbcc$ and $w_2 = cccaaccbbbb$.

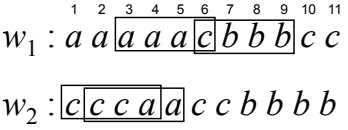

Figure 7.13: Showing two sliding windows of length $d = 4$, where $i = 3$, $\mathrm{bp}_1(i) = 6$, $k = 1$, $\mathrm{bp}_2(k) = 2$, $c_{p_l} = a$, $c_{p_r} = b$, $c_{q_l} = c$, $c_{q_r} = a$.

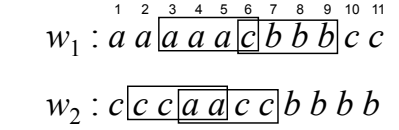

Figure 7.14: Showing two sliding windows of length $d = 4$, where $i = 3$, $\mathrm{bp}_1(i) = 6$, $k = 2$, $\mathrm{bp}_2(k) = 4$, $c_{p_l} = a$, $c_{p_r} = b$, $c_{q_l} = c$, $c_{q_r} = c$.

Suppose that the window for w_1 is now aligned at position 3 of w_1 (namely $U_3 = w_1[3..6] = aaac$). We then shift it to position $\mathrm{bp}_1(3) = 6$ (namely $U_6 = w_1[6..9] = cbbb$). For this shift of the window on w_1, we test $O(m_2)$ shifts of the window over the second string w_2, as follows.

We begin with position 1 of the other string w_2 (namely $V_1 = w_2[1..4] = ccca$), and shift the window to position $\mathrm{bp}_2(1) = 2$. See also Figure 7.13. It follows from Lemma 34 that there is no common Abelian factor during these slides. We move on to the next step.

Next, the window for w_2 is shifted from position 2 to position $\mathrm{bp}_2(2) = 4$ (namely, $V_4 =$

$w_2[4..7] = aacc$). See also Figure 7.14. It follows from Lemma 33 that there is no common Abelian factor during the slides. We move on to the next step.

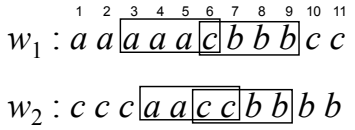

Figure 7.15: Showing two sliding windows of length $d = 4$, where $i = 3$, $\mathrm{bp}_1(i) = 3$, $k = 4$, $\mathrm{bp}_2(k) = 6$, $c_{p_l} = a$, $c_{p_r} = b$, $c_{q_l} = a$, $c_{q_r} = b$.

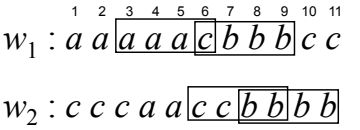

Figure 7.16: Showing two sliding windows of length $d = 4$, where $i = 3$, $\mathrm{bp}_1(i) = 3$, $k = 6$, $\mathrm{bp}_2(k) = 3$, $c_{p_l} = a$, $c_{p_r} = b$, $c_{q_l} = c$, $c_{q_r} = b$.

Next, the window for w_2 is shifted from position 4 to position $\mathrm{bp}_2(4) = 6$ (namely, $V_6 = w_2[6..9] = ccbb$). See also Figure 7.4. Since the numbers of occurrences of c on w_1 and w_2 are different and c is not equal to a or b, there is no common Abelian factor during the slides. We move on to the next step.

Next, the window for w_2 is shifted from position 6 to position $\mathrm{bp}_2(6) = 8$. See Figure 7.4. It follows from Lemma 29 that there is a common Abelian factor $(w_1[6..9], w_2[7..10])$ of length $d = 4$.

Chapter 8

Conclusion

In this thesis, we studied the problem of factorizing a string into combinatorial objects.

In Chapter 3, we considered the reversed Lempel Ziv factorization. We then presented an algorithm that computes the reversed Lempel Ziv factorization of input string of length n in $O(n \log \sigma)$ time and $O(n \log n)$ time in an online manner.

In Chapter 4, we defined the palindromic cover and the palindromic factorization. We proposed an algorithm that computes the palindromic cover in $O(n)$ time and space, and an algorithm that computes the palindromic factorization in $O(n \log n)$ time in online manner, where n is the length of input string. Moreover we showed an online algorithm computes the maximal palindromic factorization in $O(n \log n)$ time.

In Chapter 5, we introduced the diverse palindromic factorization problem, and we showed that the problem is NP-complete.

In Chapter 6, we defined the closed factorization. We proposed an algorithm that computes the closed factorization in $O(n)$ time and an algorithm that computes the longest closed factor array in $O(n \frac{\log n}{\log \log n})$ time.

In Chapter 7, we considered Abelian regularities and algorithms that efficiently compute their regularities via run length encoding. We proposed an algorithm that computes all Abelian periods in $O(mn)$ time, an algorithm that comptues all Abelian squares in $O(mn)$ time, and an algorithm that computes all longest common Abelian factors in $O(m^2 n)$ time.

Bibliography

[1] A. Al-Hafeedh, M. Crochemore, L. Ilie, J. Kopylov, W. Smyth, G. Tischler, and M. Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Computing Surveys*, 45(1):5:1–5:17, 2012.

[2] A. Alatabbi, C. S. Iliopoulos, A. Langiu, and M. S. Rahman. Algorithms for longest common Abelian factors. *International Journal of Foundations of Computer Science*, 27(5):529–544, 2016.

[3] A. Alatabbi, C. S. Iliopoulos, and M. S. Rahman. Maximal palindromic factorization. In *Proc. Prague Stringology Conference 2013 (PSC2013)*, pages 70–77, 2013.

[4] S. Alstrup and J. Holm. Improved algorithms for finding level ancestors in dynamic trees. In *Proc. 27th International Colloquium on Automata, Languages, and Programming (ICALP2000)*, volume 1853 of *Lecture Notes in Computer Science*, pages 73–84, 2000.

[5] A. Amir, A. Apostolico, T. Hirst, G. M. Landau, N. Lewenstein, and L. Rozenberg. Algorithms for jumbled indexing, jumbled border and jumbled square on run-length encoded strings. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE 2014)*, pages 45–51, 2014.

[6] A. Apostolico, D. Breslauer, and Z. Galil. Parallel detection of all palindromes in a string. *Theoretical Computer Science*, 141(1&2):163–173, 1995.

[7] A. Apostolico and M. Crochemore. Fast parallel Lyndon factorization with applications. *Mathematical Systems Theory*, 28(2):89–108, 1995.

[8] G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto. Closed factorization. *Discrete Applied Mathematics*, 212:23–29, 2016.

[9] G. Badkobeh, G. Fici, and Z. Lipták. A note on words with the smallest number of closed factors. http://arxiv.org/abs/1305.6395, 2013.

[10] G. Badkobeh, T. Gagie, S. Grabowski, Y. Nakashima, S. J. Puglisi, and S. Sugimoto. Longest common Abelian factors and large alphabets. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, pages 254–259, 2016.

[11] H. Bannai, T. Gagie, S. Inenaga, J. Kärkkäinen, D. Kempa, M. Piatkowski, S. J. Puglisi, and S. Sugimoto. Diverse palindromic factorization is NP-complete. In *Proc. 19th International Conference on Developments in Language Theory (DLT 2015)*, pages 85–96, 2015.

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

[13] K. Borozdin, D. Kosolobov, M. Rubinchik, and A. M. Shur. Palindromic length in linear time. In *Proc. 28th Annual Symposium on Combinatorial Pattern Matching (CPM2017)*, pages 23:1–23:12, 2017.

[14] S. Brlek, J.-O. Lachaud, X. Provençal, and C. Reutenauer. Lyndon + Christoffel = digitally convex. *Pattern Recognition*, 42(10):2239–2246, 2009.

[15] S. Buss and M. Soltys. Unshuffling a square is NP-hard. *Journal of Computer and System Sciences*, 80(4):766–776, 2014.

[16] K. Casel, H. Fernau, S. Gaspers, B. Gras, and M. L. Schmid. On the complexity of grammar-based compression over fixed alphabets. In *Proc. 43rd International Colloquium on Automata, Languages, and Programming (ICALP2016)*, pages 122:1–122:14, 2016.

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

[18] S. Constantinescu and L. Ilie. Fine and Wilf's theorem for Abelian periods. *Bulletin of the EATCS*, 89:167–170, 2006.

[19] M. Crochemore and L. Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.

[20] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, J. Pachocki, J. Radoszewski, W. Rytter, W. Tyczyński, and T. Waleń. A note on efficient computation of all Abelian periods in a string. *Information Processing Letters*, 113(3):74–77, 2013.

[21] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, pages 22–34, 2016.

[22] M. Crochemore, G. M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003.

[23] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.

[24] L. J. Cummings and W. F. Smyth. Weak repetitions in strings. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 24:33–48, 1997.

[25] J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Parallel RAM algorithms for factorizing words. *Theoretical Computer Science*, 127(1):53–67, 1994.

[26] M. Dumitran, F. Manea, and D. Nowotka. On prefix/suffix-square free words. In *Proc. 22nd International Symposium on String Processing and Information Retrieval (SPIRE 2015)*, pages 54–66, 2015.

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

[28] J.-P. Duval, R. Kolpakov, G. Kucherov, T. Lecroq, and A. Lefebvre. Linear-time computation of local periods. *Theoretical Computer Science*, 326(1-3):229–240, 2004.

[29] P. Erdös. Some unsolved problems. *Hungarian Academy of Sciences Mat. Kutató Intézet Közl*, 6:221–254, 1961.

[30] H. Fernau, F. Manea, R. Mercaş, and M. L. Schmid. Pattern matching with variables: Fast algorithms and new hardness results. In *Proc. 32nd Symposium on Theoretical Aspects of Computer Science (STACS2015)*, pages 302–315, 2015.

[31] G. Fici. A classification of trapezoidal words. In *Proc. 8th International Conference Words 2011 (WORDS 2011)*, Electronic Proceedings in Theoretical Computer Science 63, pages 129–137, 2011. See also `http://arxiv.org/abs/1108.3629v1`.

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

[33] G. Fici, T. Lecroq, A. Lefebvre, and É. Prieur-Gaston. Algorithms for computing Abelian periods of words. *Discrete Applied Mathematics*, 163:287–297, 2014.

[34] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, and W. F. Smyth. A note on easy and efficient computation of full Abelian periods of a word. *Discrete Applied Mathematics*, 212:88–95, 2016.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.

[36] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen. Faster longest common extension queries in strings over general alphabets. In *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM2016)*, pages 5:1–5:13, 2016.

[37] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.

[38] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. Data Compression Conference (DCC2013)*, pages 133–142, 2013.

[39] S. Grabowski. A note on the longest common Abelian factor problem. *CoRR*, abs/1503.01093, 2015.

[40] D. Hucke, M. Lohrey, and C. P. Reh. The smallest grammar problem revisited. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE 2016)*, pages 35–49, 2016.

[41] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Efficient lyndon factorization of grammar compressed text. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM2013)*, pages 153–164, 2013.

[42] T. I, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Faster Lyndon factorization algorithms for SLP and LZ78 compressed text. *Theoretical Computer Science*, 656:215–224, 2016.

[43] T. I, S. Sugimoto, S. Inenaga, H. Bannai, and M. Takeda. Computing palindromic factorizations and palindromic covers on-line. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM2014)*, pages 150–161, 2014.

[44] H. Inoue, Y. Matsuoka, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Computing smallest and largest repetition factorizations in $O(n \log n)$ time. In *Proc. Prague Stringology Conference 2016 (PSC2016)*, pages 135–145, 2016.

[45] J. Jansson, K. Sadakane, and W.-K. Sung. Compressed dynamic tries with applications to LZ-compression in sublinear time and space. In *Proc. IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS2007)*, pages 424–435, 2007.

[46] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. 12th International Symposium on Experimental Algorithms (SEA2013)*, pages 139–150, 2013.

[47] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM2013)*, pages 189–200, 2013.

[48] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. Meeting on Algorithm Engineering & Experiments (ALENEX2013)*, pages 103–112, 2013.

[49] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[50] T. Kociumaka, J. Radoszewski, and W. Rytter. Fast algorithms for Abelian periods in words and greatest common divisor queries. In *Proc. 30th Symposium on Theoretical Aspects of Computer Science (STACS2013)*, pages 245–256, 2013.

[51] T. Kociumaka, J. Radoszewski, and B. Wisniewski. Subquadratic-time algorithms for Abelian stringology problems. In *Proc. 6th International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS2015)*, pages 320–334, 2015.

[52] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS1999)*, pages 596–604, 1999.

[53] R. Kolpakov and G. Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, 410(51):5365–5373, 2009.

[54] D. Kosolobov. Computing runs on a general alphabet. *Information Processing Letters*, 116(3):241–244, 2016.

[55] S. Kreft and G. Navarro. LZ77-like compression with fast random access. In *Proc. Data Compression Conference (DCC2010)*, pages 239–248, 2010.

[56] S. Kreft and G. Navarro. Self-indexing based on LZ77. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM2011)*, pages 41–54, 2011.

[57] M. Kufleitner. On bijective variants of the Burrows-Wheeler transform. In *Proc. Prague Stringology Conference 2010 (PSC2009)*, pages 65–79, 2009.

[58] L. Levin. Universal search problems. *Problems of Information Transmission*, 9(3):115–116, 1973.

[59] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3), 2008.

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

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

[62] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science*, 410(8–10):900–913, 2009.

[63] Y. Matsuoka, S. Inenaga, H. Bannai, M. Takeda, and F. Manea. Factorizing a string into squares in linear time. In *Proc. 27th Annual Symposium on Combinatorial Pattern Matching (CPM2016)*, pages 27:1–27:12, 2016.

[64] G. Navarro and Y. Neckrich. Personal Communication.

[65] G. Navarro and Y. Neckrich. Sorted range reporting. In *Proc. 13th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT2012)*, LNCS 7357, pages 271–282. Springer, 2012.

[66] D. Okanohara and K. Sadakane. An online algorithm for finding the longest previous factors. In *Proc. 16th Annual European Symposium on Algorithms (ESA2008)*, pages 696–707, 2008.

[67] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[68] M. L. Schmid. Computing equality-free and repetitive string factorizations. *Theoretical Computer Science*, 618(7):42–51, 2016.

[69] R. Siromoney and L. Mathew. A public key cryptosystem based on Lyndon words. *Information Processing Letters*, 35(1):33–36, 1990.

[70] T. A. Starikovskaya. Computing Lempel-Ziv factorization online. In *Proc. 37th International Symposium on Mathematical Foundations of Computer Science (MFCS2012)*, pages 789–799, 2012.

[71] S. Sugimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Computing reversed Lempel-Ziv factorization online. In *Proc. Prague Stringology Conference 2013 (PSC2013)*, pages 107–118, 2013.

[72] S. Sugimoto, N. Noda, S. Inenaga, H. Bannai, and M. Takeda. Computing abelian regularities on rle strings. In *Proc. 28th International Workshop on Combinatorial Algorithms 2017 (IWOCA2017)*, 2017. To appear.

[73] G. S. Tseitin. On the complexity of derivation in propositional calculus. In A. O. Slisenko, editor, *Structures in Constructive Mathematics and Mathematical Logic, Part II*, pages 115–125. 1968.

[74] P. Weiner. Linear pattern-matching algorithms. In *Proc. 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[75] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[76] J. Yamamoto, T. I, H. Bannai, S. Inenaga, and M. Takeda. Faster compact on-line Lempel-Ziv factorization. In *Proc. 31st Symposium on Theoretical Aspects of Computer Science (STACS2014)*, 2014.

[77] C.-C. Yu, W.-K. Hon, and B.-F. Wang. Improved data structures for the orthogonal range successor problem. *Computational Geometry*, 44(3):148–159, 2011.

[78] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–349, 1977.

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