Exploring Regular Structures in Strings

Dissertation

zur Erlangung des Grades eines

Doktors der Naturwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Dominik Köppl

Dortmund

2018

Tag der mündlichen Prüfung:	*17. Juli 2018*
Dekan:	*Prof. Dr.-Ing. Gernot A. Fink*
Gutachter:	*Prof. Dr. Johannes Fischer*, und
	Prof. Dr. Shunsuke Inenaga

Acknowledgments

The first words are dedicated to those who helped and supported me for finishing this thesis.

First and foremost, I want to express my gratitude to my advisor Johannes Fischer for giving me the opportunity to work in his research group at the TU Dortmund. I am deeply grateful for his scientific guidance during the past years, for encouraging me to present my results at conferences, and for giving me the freedom to pursue my own research interests with a high degree of independence.

Second, I am grateful to Shunsuke Inenaga for hosting me during the JSPS summer program 2016, for the interesting scientific discussions during various meetings (leading to two conference contributions and one journal article), and for immediately agreeing to serve as the second referee of this thesis.

I thank Thomas Schwentick and Sven Rahmann for agreeing to serve as members of my defense committee. I want to thank Thomas Schwentick for holding the mini-workshop for theoretical computer science at the faculty, at which I could contribute several times.

I am obliged to the co-authors of my articles for sharing their experience and knowledge. My special thanks are directed to my co-authors Roland Glück, Tomohiro I, and Florian Kurpicz, who were also willing to help me polishing up this work. I also want to thank my co-author Kunihiko Sadakane for welcoming me twice as a guest at his research group at the Tokyo University.

I owe a big thanks to Jonas Charfreitag, Andre Droschinsky, Jonas Ellert, Marius Greiff, Jannik Junghänel, Bastian Kuhn, Elias Kuthe, Alexander Köppl, Bianca Markowski, Manuel Mulzer, Christopher Osthues, Arno Pasternak, Jonas Schmidt, Henning Timm, Elias Wiebelitz, and Jens Zentgraf for proofreading parts of this thesis and making valuable suggestions.

I am also grateful to my scientific student assistants Patrick Dinklage and Marvin Löbel without whom the framework tudocomp would not have been realizable. I want to thank especially our secretary Gundel Jankord and our technical administrator Helmut Henning for making paperwork as agreeable as possible.

Abstract

This thesis is dedicated to string processing algorithms and to combinatorics on words. With respect to the former, we devise Lempel-Ziv (LZ) factorization and sparse suffix sorting algorithms. With respect to the latter, we search for all distinct squares, all maximal α-gapped repeats, and all maximal α-gapped palindromes. The topics of the presented approaches are related: for instance, our results for the LZ factorization have applications for finding all distinct squares, whereas our sparse suffix sorting algorithm profits from finding gapped repeats. The results can contribute to tools for data compression, for text indexing, and for the analysis of biological sequences.

Given a text T of length n whose characters are drawn from an integer alphabet of size σ, we obtain the following results within the RAM model:

LZ factorizations. We devise algorithms computing Lempel-Ziv-77 (LZ77) and Lempel-Ziv-78 (LZ78) in $\mathcal{O}(n/\epsilon) = \mathcal{O}(n)$ time while taking $\min(\mathcal{O}(n \lg \sigma), (1 + \epsilon)n \lg n + \mathcal{O}(n))$ bits of working space, where ϵ with $0 < \epsilon \leq 1$ is a selectable constant trade-off parameter. We also show that the LZ78 factorization can be computed with a Las Vegas algorithm in $\mathcal{O}(n)$ time with $\mathcal{O}(z \lg(\sigma z))$ bits of working space, where z is the number of LZ78 factors. Previous algorithms use more memory or need superlinear time.

Applications of the LZ77 factorization. We revisit the longest previous factor (LPF) table, which is traditionally stored in an array of $n \lg n$ bits. We propose a succinct representation of the LPF table taking $2n + o(n)$ bits, and compute the LPF table with an adaptation of our LZ77 factorization algorithms, running in $\mathcal{O}(n \lg^{\epsilon}_{\sigma} n)$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space, or in $\mathcal{O}(n)$ time with $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of working space. Having the LPF table at hand, we devise an algorithm computing all distinct squares in $\mathcal{O}(n)$ time with twice the amount of the working space, i.e., with either $2(1 + \epsilon)n \lg n + \mathcal{O}(n)$ or $\mathcal{O}(n \lg \sigma)$ bits, for computing the LPF table. Additionally, we propose the first algorithm computing all distinct squares *online*. The online algorithm runs in $\mathcal{O}(n \lg^2 \lg n / \lg \lg \lg n)$ time.

Sparse Suffix Sorting. We propose an online algorithm computing the sparse suffix array and the sparse longest common prefix array of the text T in $\mathcal{O}(c(\sqrt{\lg \sigma} + \lg \lg n) + m \lg m \lg n \lg^* n)$ time with $\mathcal{O}(m)$ words of space under the premise that the space of T is rewritable, where $m \leq n$ is the number of

suffixes to be sorted (provided online and arbitrarily), and c is the number of characters with $m \leq c \leq n$ that must be compared for distinguishing the designated suffixes. This is the first *deterministic* approach with a working space limit of $\mathcal{O}(m)$ words and non-trivial running time. Previous approaches are randomized or impose restrictions on the selected suffixes.

All maximal α-gapped repeats and palindromes. We show that the number of all maximal α-gapped repeats and the number of all maximal α-gapped palindromes are at most $3(\pi^2/6 + 5/2)\alpha n$ and $7(\pi^2/6 + 1/2)\alpha n - 3n - 1$, respectively. Subsequently, we present an algorithm finding all maximal α-gapped repeats in $\mathcal{O}(\alpha n)$ time. This is the first algorithm that runs in linear time independently of σ, as previous results with the same running time assumed σ to be constant.

Contents

Chapter 1

Introduction

From rainbows, river meanders, and shadows to spider webs, honeycombs, and the markings on animal coats, the visible world is full of patterns that can be described mathematically.

— John A. Adam [3]

Handling large texts spans a wide range of problems. Most prominent examples are (1) *text compression* to store and transfer textual data, (2) the *indexing* of full texts to search in textual data, and (3) the analysis of *characteristics* in textual data (e.g., evaluating biological data). Naïve approaches tackling these problems fail for textual data in general because they tend to be too slow or require memory that is orders of magnitude larger than the data. To still handle large data sets in reasonable time and memory, we need to shift our attention to solutions whose performance scales well with the problem instance. These solutions need (a) to work in linear or near-linear time to have guarantees about their scalability regarding the speed, and (b) to use data structures with small memory footprints to support working with large textual data within the available memory resources.

The key technique of our solutions is based on finding and exploiting *regular structures*. Regular structures like repeats and palindromes help us to pave new ways to (1) compress texts, (2) speed up text comparisons, and (3) characterize texts. Regular structures are ubiquitous: They are found in biological data like deoxyribonucleic acid (DNA) or ribonucleic acid (RNA) sequences, natural languages or programming languages. For instance, the common German tongue-twister[1]

fischers	fritz	fischt	frische	fische	frische	fische	fischt	fischers	fritz
1	2	3	4	5	6	7	8	9	10

is rich in regular structures: the re-occurrences of `fischers fritz` and `fischt` are called *gapped repeats*, the substring consisting of the two consecutive occurrences of `frische fische` is called a *square*, i.e., a square is a gapped repeat with a gap of length zero. The first occurrence and the repeated occurrence of a gapped repeat are called its *left* and *right arm*, respectively. By means of this example text, we intend to highlight some concepts that we study in this thesis:

[1] We enumerated the words to ease the following explanations.

(1) A compressor can apply the Lempel-Ziv-77 (LZ77) factorization [246] to remove all re-occurrences. The word-based LZ77 factorization is computed by scanning the text from left to right. Starting at the first word ($i = 1$), we process the text as follows: Given that we are at the i-th word, we replace the right arm of the longest gapped repeat starting at the i-th word with a reference to its left arm, and move to the $(i + \ell)$-th word, where ℓ is the number of words covered by the left arm. If the left arm starts with the j-th word, the reference is stored as a pair (j, ℓ), which is also an instruction for the decompressor to substitute the tuple with the j-th word and its $\ell - 1$ succeeding words. If there is no gapped repeat starting at the i-th word, we move to the $(i + 1)$-th word and recurse. For our tongue-twister the output is `fischers fritz fischt frische fische (4,2) (3,1) (1,2)`, in which the right arms of all previously described gapped repeats are replaced by references.

(2) An index for substring comparisons could maintain a dictionary that addresses each stored string by a name. The index replaces both arms of a gapped repeat with a name, and stores the characters of an arm along with its name in the dictionary. For instance, we could give the names `A`, `B` and `C` to the substrings `fischers fritz`, `fischt`, and `frische fische`, respectively, and replace all occurrences of these substrings with their respective names.

This gives us a new text $T' :=$ `ABCCBA`. Since we can restore the original text with the dictionary and T', we can discard T. For highly repetitive texts, storing the dictionary and T' is more space efficient than keeping T. A query such as comparing the suffixes starting at the fourth and the sixth word (i.e., starting at the third and fourth replaced substring) can now be performed quicker by looking at the names instead of the words themselves. Queries can be answered with an index data structure like the *suffix tree* even more efficiently. The suffix tree for T'`$` is given in Fig. 1.1. We appended the delimiter `$` to ensure that a suffix starting at position i is represented by a leaf with number i.

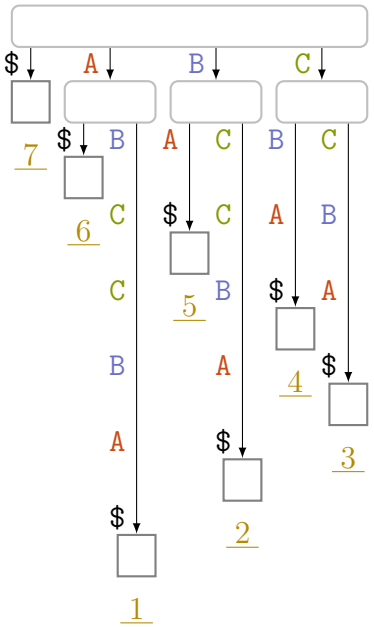


Fig. 1.1: Suffix tree of $T' =$ `ABCCBA`.

We also stipulate that the delimiter `$` is the lexicographically smallest character. It can appear only at the end of a text. In the suffix tree, we

Compression Scheme	Regularity
LZ77 (Sect. 3.4)	Gapped Repeats
Reversed LZ77 (Sect. 3.6)	Gapped Palindromes
LCPcomp [69, Sect. 3.2]	LCP Array Histogram
Huffman code [126]	Entropy H_0
PPM [51]	Entropy H_k

Fig. 1.2: Compression schemes taking advantage of specific regularities.

sort the children of each node lexicographically, and label each leaf with the starting position of its respective suffix.

(3) More can be seen when stepping down from word level to character level: The text contains mainly occurrences of `fri`, `isch` and `fisch`, which are interspersed by some characters. All these occurrences are part of gapped repeats. The left and right arms of some of these gapped repeats are inside the respectively left and right arm of other gapped repeats. We call those gapped repeats *maximal* whose arms are not contained in the arms of another gapped repeat, For instance, the re-occurrence `fischt` is not maximal, whereas the re-occurrence `fischt␣f` is maximal.

We also study *periodic substrings*. A periodic substring is a consecutive repetition of a string. Since periodic substrings contain at least two consecutive occurrences of a string, they are a generalization of squares. For instance, the substring `frische␣fische␣frische␣fische␣` is both a square and a periodic substring, whereas `␣frische␣fische␣frische␣fische␣f` is only a periodic substring. The latter periodic substring is *maximal*, i.e., it can be extended neither to its left nor to its right to a longer periodic substring (in our case, neither the preceding `t` nor the succeeding `i` do fit). We call such a maximal periodic substring also a *run*.

Apart from that, the string $T' =$ `ABCCBA` defined in (2) is a *palindrome*. Similar to periodic substrings, we say that a palindrome is *maximal* if it cannot be extended outwards (like we could extend the occurrence of `BCCB` in T' to `ABCCBA`). We profit from the restriction to maximal repetitions or maximal palindromes in that their number in a text T is at most linear in the length of T, whereas there can be quadratic many squares or palindromes found in T. Combinatorial results providing upper bounds on the number of a regular structure help us in attaining an upper bound on the running time of algorithms finding all occurrences of this regular structure.

1.1 Joining the Dots

The keynote of the introduction of this chapter is to think of regular structures as redundancies of the text. Removing redundancies is the key concept of compression schemes that substitute redundant parts to gain compression. Put differently, regular structures give evidence of the compressibility of a string, whereas compression algorithms classify regular structures as redundant parts of the text. As can be seen in Fig. 1.2, analyzing regular structures and devising compression algorithms goes hand in hand. Figure 1.3 gives an overview of the interaction of the topics that we focus on within this thesis.

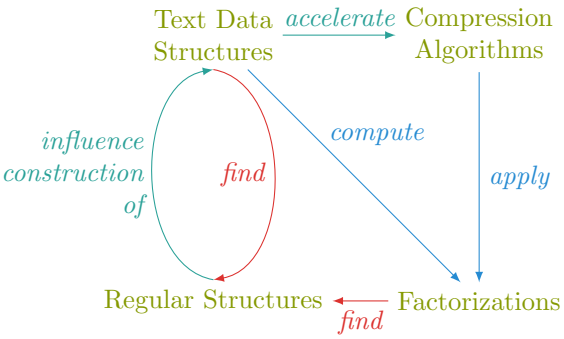


Fig. 1.3: Interaction of the topics within our focus.

It would go beyond the scope of this thesis to give a thorough review of all regular structures (for that see, e.g., [18, 63, 178]) or of the importance of regular structures. Instead, we briefly sketch, for a text of length n whose characters are drawn from an integer alphabet of size σ, the history and importance of two fundamental tools for working with regular structures: The suffix tree and the LZ77 factorization. The suffix tree has a long story of advancements, starting with the construction algorithms of Weiner [241], McCreight [187], and Ukkonen [236], each with a running time of $\mathcal{O}(n \lg \sigma)$, which was improved to $\mathcal{O}(n)$ time later by Farach-Colton et al. [75]. Gusfield [120] devoted most of his book to the usefulness of suffix trees, giving plenty of examples on how to find regular structures with the suffix tree. He also showed how to compute the LZ77 factorization with the suffix tree in $\mathcal{O}(n \lg \sigma)$ time. However, the suffix tree is traditionally stored in a pointer-based tree structure, which takes $\mathcal{O}(n \lg n)$ bits of memory. This memory footprint makes pointer-based suffix trees unfavorable for large datasets. Just recently, Munro et al. [190] presented a suffix tree construction algorithm using $\mathcal{O}(n \lg \sigma)$ bits of working space. The algorithm constructs an $\mathcal{O}(n \lg \sigma)$-bits representation of the suffix tree in linear time. The drawback is that some operations, like querying the suffix number of a leaf, are by a logarithmic factor slower than in the pointer-based representation. Another representation [2] of the suffix tree combines the suffix array with (succinct) support data structures. With this representation Abouelhoda et al. [2] can compute the LZ77 factorization. For that, they need the longest common prefix (LCP) array. The interesting connection between the LCP array and the LZ77 factorization is that the LCP array already stores the lengths of all LZ77 factors. This connection was observed by Crochemore et al. [62]. They studied the longest previous factor (LPF)

table [58, 98] whose i-th entry stores the length of longest substring starting before i that is a prefix of the suffix starting at text position i, i.e., the i-th entry is the length of an LZ77 factor that would start at the i-th text position (cf. Sect. 3.4.4). Their study lead to the observation that the LPF table is a permutation of LCP array [62, Prop. 2].

Back to the suffix tree representations, a question is whether we can compute the LZ77 factorization in *linear* time in reasonable space. As we will later see, we can answer this question affirmatively by presenting two algorithmic solutions working with (a) the compressed suffix tree of Munro et al. [190] within $\mathcal{O}(n \lg \sigma)$ bits and (b) the suffix array enhanced with support data structures within $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits for an arbitrary constant $\epsilon > 0$, respectively. Having efficient algorithms computing the LZ77 factorization available, we present its usefulness for finding regular structures by giving an application on computing all distinct squares.

To close the cycle between regular structures and string algorithms, we emphasize that regular structures can help devising string algorithms efficiently. An interesting problem, for instance, is whether we can take advantage of regular structures like gapped repeats to accelerate suffix comparisons. Suppose that we want to sort m suffixes of a text T of length n while using only $\mathcal{O}(m)$ words as working space. This can be done by a plain string sorter that treats the m suffixes as m strings (without considering that the suffixes overlap). On the one hand, the string sorter is best at sorting the m suffixes if they pairwisely share a longest common prefix of constant length. In that case, the string sorter can distinguish two suffixes in constant time. For instance, an in-place linear time integer sorting algorithm (e.g., Lemma 2.7) can sort the suffixes in $\mathcal{O}(m)$ time using $\mathcal{O}(m)$ words of space to store the order of the suffixes. On the other hand, the string sorter runs in $\Omega(nm)$ time to sort the first m suffixes if T is a run of the same character (i.e., $T = \mathtt{a}^n$ for a character $\mathtt{a}$). In Chapter 4, we show how to prevent the string sorter from running in quadratic time with a data structure built on gapped repeats.

1.2 Our Results

Highlighting three different aspects of working with regular structures in the opening of this introduction was not unintentional. In this thesis, we tackle problems that can be categorized into these three aspects, namely:

(1) We devise algorithms computing the Lempel-Ziv-77 (LZ77) [246] and Lempel-Ziv-78 (LZ78) [247] factorizations,

(2) we construct the sparse suffix array online, and

(3) we devise an upper bound on the number of all maximal α-gapped repeats and palindromes, and show how to compute the sets

- of all maximal α-gapped repeats,
- of all maximal α-gapped palindromes, and
- of all distinct squares.

We devised our algorithms within the RAM model. We assume that an algorithm receives an input text (of length n) whose characters are drawn from an integer alphabet (of size σ). Except for the hashing techniques presented in Sect. 3.8, all algorithms work *deterministic*.

Since the computation of all distinct squares depends on the LZ77 factorization, we felt that this result fits better in the chapter dedicated to the Lempel-Ziv (LZ) factorizations. Having said that, the structure of this thesis is as follows:

1.2.1 LZ Factorizations

In Chapter 3, we provide new algorithms for computing the LZ factorizations. The LZ factorizations partition a text into substrings, which are called *factors*. Each factor refers either (LZ77) to a previous text position or (LZ78) to a previous factor. These references are selected *greedily*, i.e., the referred position of an LZ77 factor F is the starting position of the *longest* substring starting before F that is equal to F, whereas the referred factor of an LZ78 factor F is the factor that has the *longest* common prefix with F among all other factors preceding F. Both factorizations allow us to represent each factor as a pair of integers containing (LZ77) the referred position and the factor length, or (LZ78) the referred factor and an additional character. These pairs of integers can be encoded in a compressed file. The crux is to find the referred positions and the referred factors. For instance, scanning the text naïvely to find the referred position for each factor takes quadratic time. Although introduced back in 1977 and 1978, research is still active on this topic, yielding incremental improvements in computing the LZ factorizations faster and more memory efficiently. In this line of research, we present algorithms computing the LZ77 (Sect. 3.4) and the LZ78 (Sect. 3.7) factorization in $\mathcal{O}(n)$ time while taking $\min(\mathcal{O}(n \lg \sigma), (1+\epsilon)n \lg n + \mathcal{O}(n))$ bits of working space, where ϵ with $0 < \epsilon \leq 1$ is a selectable constant. These are the first algorithms taking space that is at most linear in the number of *bits* of the input text size. Previous results achieved non-linear time or work with more memory. The currently best linear time algorithms for computing the LZ77 factorization are by Kärkkäinen et al. [148] and Goto and Bannai [116]. The former group of authors presented an algorithm using $2n \lg n$ bits of memory. The latter group uses a single array with $n \lg n$ bits and support data structures with $\mathcal{O}(\sigma \lg n)$ bits. For $\sigma = \Omega(n)$, we can select a sufficient small ϵ such that our solution using $(1+\epsilon)n \lg n + 2n$ bits (Thm. 3.11) takes less space: We set $\epsilon := \min(1, \sigma/n - 2/\lg n)$, which is positive if $\sigma/n > 2/\lg n$ (this inequality holds for sufficiently large σ). With this ϵ our working space becomes $n \lg n + \sigma \lg n$ bits (if $\sigma \leq n + 2n/\lg n$) or $2n \lg n$ bits (if $\sigma > n + 2n/\lg n$). Our algorithm runs in $\mathcal{O}(n/\epsilon) = \mathcal{O}(n^2/\sigma) = \mathcal{O}(n)$ time.

However, we mostly deal with small alphabet sizes when working with the LZ77 factorization in practice. Typical large texts with small alphabet sizes are DNA sequences. Suppose that we want to compute the LZ77 factorization on a collection of 100 human genomes (to illustrate our approach with concrete numbers). A human genome has approximately $3 \cdot 10^9$ base pairs, where each base is of one of four different types (A, C, G, or T) such that the human genome can be represented as a string whose characters are drawn from an alphabet of size four. By representing each base with two bits, a human genome can be stored in $6 \cdot 10^9$ bits, or roughly 715 MiB. Consequently, a collection of 100 human genomes can be represented as a text of size 71,500 MiB $\approx$ 69.8 GiB with $\sigma = 4$. The LZ77 factorization algorithm of Gusfield [120] using a pointer-based suffix tree representation is unattractive with respect to the memory footprint. For instance, the implementation by Kurtz [169] is considered one of the most efficient suffix tree implementations. However, this implementation already takes 80 bits on average and up to 160 bits per character. As Fig. 1.4 shows, there are several compressed suffix tree (CST) implementations available, whose memory footprints range from 4 up to 17 bits per character. Our LZ77 factorization algorithm uses, additionally to the CST, at most $3n + z \lg n + o(n)$ bits (Thm. 3.13), where z is the number of LZ77 factors. For the second term, we can assume that $z \lg n \leq n \lg \sigma = 2n$ for a compressible DNA sequence (like in our case[2]). The $o(n)$ term is due to a rank-support that usually takes a fraction of n [196]. Within these assumptions, we need at most $6n$ bits on top of the CST representation, i.e., between 10 and 22 bits per character on average. This yields a range between 349 GiB and 803 GiB as an estimate for the maximal memory consumption of our LZ77 algorithm. The approach of Goto and Bannai [116] needs $n \lg n + \mathcal{O}(\sigma \lg n) \geq n \lg n$ bits of memory, or at least $39n$ bits with $\lceil \lg n \rceil = 39$. Consequently, they need 1362 GiB of memory *at least*. According to these numbers, our memory requirement is less than 26%–59% of the memory needed by Goto and Bannai [116]. As pointed out by Abeliuk et al. [1, Sect. 1], the memory requirement of a CST representation can be much less (down to 0.6 bits per character) in the case of repetitive text collections like a collection of human genomes. A downside of our approach is that current CST implementations need more space during their *construction* such that a practical evaluation of the memory consumption does not lead to satisfiable results (yet).

Up to a term of $\mathcal{O}(n)$ bits, our LZ77 and LZ78 factorization algorithms work within the same spaces. For LZ78, it is possible to further improve the memory footprint of the LZ78 factorization when permitting randomization: In Sect. 3.8, we present a randomized algorithm computing the LZ78 factorization in $\mathcal{O}(n)$ expected time with $\mathcal{O}(z \lg(\sigma z))$ bits of working space for the same text, where z is the number of LZ78 factors. This variant is actually practical as we evaluate it within a framework [69] using a dynamic trie data structure for computing the

[2] For instance, see [239] for an approach compressing multiple human genomes with LZ77.

Bits	Ref.
4–6	Russo et al. [215]
8–12	Abeliuk et al. [1]
13–16	Abeliuk et al. [1]
13–17	Gog and Ohlebusch [110]

Fig. 1.4: CST implementations with an estimate of the needed space (average number of bits per character). There is a trade-off between the memory footprint and the practical speed (not shown here).

LZ78 factorization. For LZ78-compressible texts, we can experimentally validate that our algorithm needs *less bits* than the input.

Finally, we consider variants of the LZ77 factorization: (a) the classic-LZ77 factorization, in which each factor introduces an additional character like LZ78, (b) the LZ77 factorization without overlaps, and the reversed LZ77 factorization (c) with and (d) without overlaps (see the last paragraphs of Sects. 3.4.2 and 3.4.3 for the first variant and Sect. 3.6 for the other variants).

1.2.2 Applications of the LZ77 Factorization

In an intermezzo between the standard LZ77 factorization (Sect. 3.4.3) and its variants (Sect. 3.6), we give applications to the results achieved so far: We recall the LPF table [58, 98] in Sect. 3.4.4, which is traditionally stored in an array with $n \lg n$ bits. We propose a succinct representation taking $2n + o(n)$ bits, and compute the LPF table in this representation with an adaptation of our LZ77 factorization algorithms. We present two algorithms computing the LPF table (a) in $\mathcal{O}(n \lg^{\epsilon}_{\sigma} n)$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space, or (b) in $\mathcal{O}(n)$ time with $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of working space, respectively.

As an application to the LPF table, we focus on the computation of all distinct squares in Sect. 3.5. It is known that there can be $\mathcal{O}(n^2)$ occurrences of squares in a text of length n (consider $\mathtt{a}^n$). Common research focuses on counting (a) those squares that do not contain another square [244], and (b) all distinct squares, i.e., to count only one occurrence of each square present in the text. For the latter, the best known algorithm finding all distinct squares runs in linear time while taking $\mathcal{O}(n)$ words [65]. Unsatisfied with this space bound, we present an alternative algorithm with refined space requirements: Having the LPF table at hand, we present an algorithm finding all distinct squares in $\mathcal{O}(n)$ time with twice the amount of the working space needed by one of the two aforementioned algorithms computing the LPF table.

If we allow $\mathcal{O}(n)$ words of space, we can perform other operations with (variants of) this algorithm: We present (a) an *online* algorithm computing all distinct squares in $\mathcal{O}(n \lg^2 \lg n / \lg \lg \lg n)$ time, and show that the set of all distinct squares can be used for (b) finding the squares common to all strings of a set with a total string length of n, and (c) computing the tree topology of the minimal augmented suffix tree (MAST) [11], while spending $\mathcal{O}(n)$ time for the last two problems. Computing the tree topology of the MAST was previously

achieved in $\mathcal{O}(n \lg n)$ time [38]. The MAST was introduced by Apostolico and Preparata [11] for retrieving the number of *non-overlapping* occurrences of a pattern in the text.

1.2.3 Sparse Suffix Sorting

In Chapter 4, we study how string sorting can be accelerated by taking advantage of regular structures. We focus on the special topic of sorting a *subset* of suffixes of a single string *online*. This is a variant of the common problem of computing the suffix array storing the lexicographic order of *all* suffixes. It is motivated by the fact that it is often not necessary to have every entry of the suffix array available. The idea is to compute only those suffix array entries that are actually requested; we call such a suffix array *sparse*. However, in most cases, it is not clear in advance which positions are actually requested. An ideal algorithm would build a dynamic sparse suffix array that supports adding a suffix *online*. Here, we propose such an algorithm: We can compute m positions of the suffix array, i.e., the order of m suffixes, in $\mathcal{O}(c(\sqrt{\lg \sigma} + \lg \lg n) + (m \lg m \lg n \lg^* n))$ time with $\mathcal{O}(m)$ words of additional working space to the text, where c is the *distinguishing prefix size*, i.e., the number of characters that are necessary to distinguish the m suffixes. We underline that the running time is *sublinear* when $c = o(n/(\sqrt{\lg \sigma} + \lg \lg n))$ and $m \lg m = o(n/ \lg n \lg^* n)$. A feature of this algorithm is that these m entries can be chosen *online*, in any given order. The algorithm works in the *restore model*, where the algorithm is allowed to use the text space itself as a working space, i.e., it is allowed to overwrite the text under the premise that it can *restore* the original text. Overall, this approach is up until now the best known *deterministic* algorithm for computing the sparse suffix array that supports adding *arbitrary* text positions. Previous deterministic approaches restrict the selection of the suffixes, need space linear in n, or run in $\Omega(nm)$ time.

We describe our approach in Sect. 4.4 with an abstract data type that (a) answers longest common extension (LCE) queries and (b) is mergeable. As representations of this abstract data type, we propose the hierarchical stable parsing (HSP) tree in Sect. 4.3 and its variant, the truncated HSP tree, in Sect. 4.5. We parametrize the latter with a trade-off parameter τ with $1 \leq \tau \leq n$ such that it answers an LCE query $\ell := \mathsf{lce}(i, j)$ for two text positions i and j with $1 \leq i, j \leq n$ in $\mathcal{O}(\lg^* n(\lg(\ell/\tau) + \tau^{\lg 3}/ \log_\sigma n))$ time. It takes $\mathcal{O}(n/\tau)$ words of space. We can build it in $\mathcal{O}(n(\lg^* n + (\lg n)/\tau + (\lg \tau)/ \log_\sigma n))$ time with additional $\mathcal{O}(\max(n/ \lg n, \tau^{\lg 3} \lg^* n))$ words of space during construction. The HSP tree is actually a slightly modified version of the edit sensitive parsing (ESP) tree [55]. However, the ESP tree has a flaw, which we unveil in Sect. 4.2.4. This flaw invalidates upper bounds claimed in several research papers (e.g., [54, 100, 186, 227, 228]). Fortunately, we can verify that our HSP tree is not affected by it (Sect. 4.3.1).

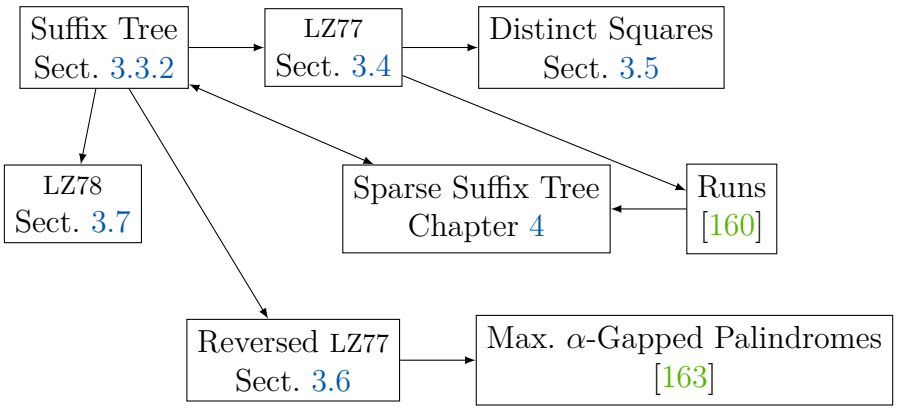


Fig. 1.5: Influence of the different techniques studied in this thesis.

1.2.4 Gapped Repeats and Palindromes

In Chapter 5, we deal with a generalization of squares and palindromes: α-gapped repeats and α-gapped palindromes. We know that the number of squares and palindromes in a string can be $\mathcal{O}(n^2)$, but the number of all runs and maximal palindromes is bounded by $\mathcal{O}(n)$. However, the set of all runs and maximal palindromes is rather tiny, and one may question whether there are other regular structures with interesting bounds. Generalizations motivated by the study of palindromic sequences of biological origin are maximal α-gapped repeats and palindromes, for which we show that the maximum number of them is tightly bounded by $\Theta(\alpha n)$ for a real number $\alpha \geq 1$ (Sect. 5.3). This result was also independently obtained by Crochemore et al. [66]. However, their analysis is restricted to gapped repeats, and yields an upper bound in order of n and α, whereas we can give concrete upper bounds, namely: There are at most $3(\pi^2/6+5/2)\alpha n$ many maximal α-gapped repeats and $7(\pi^2/6+1/2)\alpha n - 3n - 1$ many maximal α-gapped palindromes.

Accompanied by this combinatorial result, we devise an algorithm finding all maximal α-gapped repeats in $\mathcal{O}(\alpha n)$ time, obtaining an optimal result with respect to the worst case running time (Sect. 5.4). This is an improvement to the best previous approaches assuming σ to be constant [66, 229].

1.2.5 The Big Picture

As Fig. 1.5 connotes, all three parts are related: Having knowledge of the order of the suffixes and their longest common prefixes, we can compute suffix tree representations (Sect. 3.3.2) that help us computing the LZ factorizations (Sects. 3.4 and 3.7). The LZ77 factorization is the main tool for computing all distinct squares (Sect. 3.5) and all runs [160]. Knowing the runs in a string, we can accelerate LCE computation and sparse suffix array computation (Chapter 4), which can be used to compute the order of the suffixes.

1.3 Publications Contributed to this Thesis

This thesis depends on the contents of several publications. We list these publications grouped by chapter. Each listed reference is accompanied with a list of sections that are based on the respective reference. References without such a list are used as the foundation of the entire respective chapter. The names of authors are sorted in each reference alphabetically. As mandatory by the graduation exam regulations, the author appends a brief description of his contribution to each article. We start with Chapter 3, which is based on

- J. Fischer, T. I, D. Köppl, and K. Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018 [92] — Sections 3.4.2, 3.4.3, 3.7.2 and 3.7.3.
  This article consists of the following conference papers:
  - D. Köppl and K. Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12, 2016 [167] — Sections 3.4.2 and 3.7.2.

    The ideas of this contribution were conceived by the authors together during a research stay at the University of Tokyo.
  - J. Fischer, T. I, and D. Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184, 2015 [89] — Sections 3.4.3 and 3.7.3.

    All parts were developed by the authors together.

- H. Bannai, S. Inenaga, and D. Köppl. Computing all distinct squares in linear time for integer alphabets. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 22:1–22:18, 2017 [22] — Sections 3.4.4 and 3.5.

  The ideas of this contribution were developed in a joint effort during a research stay at the Kyushu University. The work on the online algorithm was initiated to answer a question posed by Thomas Schwentick at the 30th mini-workshop for theoretical computer science at the TU Dortmund.

- J. Fischer and D. Köppl. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 191–207, 2017 [87] — Section 3.8.

  The results were discussed together as part of many iterations in the algorithmic engineering progress on refining the described hash tables practically. The implementation has become part of tudocomp [69].

- P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017 [69] — Sections 3.3 and 3.8.

The ideas of the LZ78U algorithm described in the paper (which is not part of this thesis) were conceived during the stay of D. Köppl at the University of Tokyo. The $\mathcal{O}(n \lg n)$-time algorithm LCPcomp is the result of the Bachelor thesis of P. Dinklage, which was supervised by J. Fischer and D. Köppl. The described framework tudocomp is mainly implemented and maintained by P. Dinklage and M. Löbel.

Chapter 4 is based on

- J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting in the restore model. Submitted, preprint available at *ArXiv* [91].
  This preprint is based on the conference paper
  - J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting on rewritable texts. In *Proc. LATIN*, volume 9644 of *LNCS*, pages 483–496, 2016 [90].

  The initial idea on this topic was conceived by T. I. All three authors worked on the conference paper [90] together. Later, while D. Köppl was extending the contents for a journal version, he discovered a discrepancy in the referenced material [55] on which the results for the ESP trees in [90] are based. This discovery lead (a) to the counter examples showing the aforementioned flaw of the ESP trees and (b) to a thorough analysis of ESP and HSP trees. All proofs were worked out by the authors together.

Chapter 5 is based on

- P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes. *TOCS*, 62(1):162–191, 2018 [108].
  This article is based on the conference paper
  - P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Efficiently finding all maximal α-gapped repeats. In *Proc. STACS*, volume 47 of *LIPIcs*, pages 39:1–39:14, 2016 [107].

  The combinatorial problem of whether there are $\mathcal{O}(\alpha n)$ maximal α-gapped repeats was posed by M. Crochemore during Stringmasters 2015 held in Warsaw. A preliminary version proving an upper bound of $\mathcal{O}(\alpha n)$ on the number of all α-gapped repeats and palindromes was conceived by T. I and D. Köppl (see [108, Lemma 7] for an easy proof of the $\mathcal{O}(\alpha n)$ upper bound on the number of all maximal α-gapped repeats). The authors could refine their result with S. Inenaga to what the combinatorial section of [107, 108] presents. The algorithmic part is an adaptation of an already existing algorithm [70]. This adaptation was initiated by P. Gawrychowski and F. Manea. It was worked out during a stay of D. Köppl at the University of Kiel while participating at the conference *WORDS* 2015.

- T. I and D. Köppl. Improved upper bounds on all maximal α-gapped repeats and palindromes. Submitted, preprint available at *ArXiv* [129] – Section 5.3.

  The ideas of this contribution were conceived by the authors after discovering that the point analysis described in [107] could be easily improved.

1.4 Style Policies

We generally favor complying to the notation used in literature, but sometimes make exceptions to ensure that the entire work is written in a unified style. Exceptions are that the community on string combinatorics has the convention to call strings *words*, and prefers the variables u, v, w for strings prior to S, T, etc. (which we use).

Long proofs may contain claims whose proofs are interesting by themselves. These claims receive their own proof, a so-called *sub-proof* within the (long) proof. A sub-proof is finished with a filled square (■) unlike a normal proof ending with a hollow square (□).

All tables and figures with a caption are labeled as figures. Figures are sorted chronologically by the time they are addressed in the text. Exceptions are a few figures in landscape format like Figs. 3.48, 3.50 and 4.45 to 4.47. Some of these figures are put at the end (see Sects. 3.10 and 4.8) of their respective chapters to prevent disturbing the text flow. A row with label i in a tabular figure displays an enumeration of natural numbers starting with one.

We prefer structuring the results of this thesis based on the needed tools (instead of grouping the results by topic). For instance, this stands out when considering the placement of the computation of all distinct squares (Sect. 3.5) within Chapter 3 dealing with LZ factorization algorithms.

For citations, we opted for citing a later published journal version (in case that we are aware of it) instead of a conference version (or even of a preprint). One reason is that the contents in a journal version are more mature and complete, based on the fact that most journals have no page restriction, but a more rigid revision system. A small downside is that comparing the publications by their dates can be misleading, as the review process of journals usually takes more time than a conference publication.

Chapter

2

Preliminaries

Certain string algorithms that were generally deemed to be irrelevant to biology just a few years ago have become adopted by practicing biologists in both large-scale projects and in narrower technical problems. Techniques previously dismissed because they originally addressed (exact) string problems where perfect data *were assumed have been incorporated as* components *of more robust techniques that handle imperfect data.*

— Dan Gusfield [120]

Throughout this thesis, we stick to the set of notations and definitions that we introduce in the following. We also recall results that are (a) well-known in the string community and (b) relevant for multiple following chapters. We refer to the books [120, 178, 194] for a comprehensive introduction to string algorithms and combinatorics on words.

2.1 Basic Notation

We write $x \coloneqq y$ or $y =: x$ to define a variable or constant x with value y. The symbol $\leftarrow$ is used to assign a value to a variable already defined. For instance, we favor writing $i \leftarrow i + 1$ for incrementing the variable i by one, since writing $i = i + 1$ (or even $i \coloneqq i + 1$) can be confusing in a more complex context.

In the pseudo code listings of our algorithms, we write incr i for $i \leftarrow i + 1$. We use the instructions break and continue inside loops. We write break to exit the innermost surrounding loop immediately, whereas we write continue to (a) jump back to the beginning of that loop and (b) to subsequently continue with the next iteration.

We denote the set of all *positive* natural numbers, the set of all integers, and the set of all real numbers, with $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$, respectively.

The functions lg and $\log_x$ denote the logarithm to the base two and the logarithm to the base x for a real number x, respectively. For convenience, we assume that $\lg n = \lceil \lg n \rceil$ for a natural number n. We say that a function $f : \mathbb{N} \to \mathbb{N}$ is *near-linear* if $f(n) = \mathcal{O}(n \lg^{\mathcal{O}(1)} n)$.

2.2 Model of Computation

Our computational model is the word RAM model [9] with word size $\Omega(\lg n)$. Accessing a word costs $\mathcal{O}(1)$ time. A thorough definition of the word RAM model with a discussion about its practical relevancy is given by Hagerup [122, Sect. 2].

We deal with three different types of alphabets: (a) constant-size alphabets, (b) *integer alphabets* whose characters are integers that fit into a constant number of words, and (c) finite alphabets.

We analyze both *deterministic* and *randomized* algorithms in this thesis. There are two special types of randomized algorithms we focus on: A *Monte Carlo* algorithm is a randomized algorithm with *expected* running time and an output that is *expected* to be correct. A *Las Vegas* algorithm differs from a Monte Carlo algorithm in that its output is *always* correct.

2.3 Intervals

A (real) *interval* $\mathcal{I} = [b, e] \subset \mathbb{R}$ for $b, e \in \mathbb{R}$ is the set of all real numbers $i \in \mathbb{R}$ with $b \leq i \leq e$. We write $[b, e)$, $(b, e]$ or (b, e) if e, b, or both values are not included in the interval, respectively.

A special kind of intervals are *integer intervals* $\mathcal{I} = [b \,..\, e]$, where $\mathcal{I}$ is the set of consecutive integers from $b \in \mathbb{Z}$ to $e \in \mathbb{Z}$, for $b \leq e$. Given an integer interval $\mathcal{I}$, $\mathsf{b}(\mathcal{I})$ and $\mathsf{e}(\mathcal{I})$ denote the beginning and the end of $\mathcal{I}$, respectively, i.e., $\mathcal{I} = [\mathsf{b}(\mathcal{I}) \,..\, \mathsf{e}(\mathcal{I})]$. We write $|\mathcal{I}|$ to denote the length of $\mathcal{I}$; i.e., $|\mathcal{I}| = \mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$. For $b > e$ we stipulate that $[b \,..\, e] = \emptyset$.

Given numbers $i_1, \ldots, i_j$ with $j \geq 2$ whose values are between two integers b and e with $b \leq e$, we write $b \leq i_1, \ldots, i_j \leq e$, or $b \leq i_1, i_2 \leq e$ in case $j = 2$.

2.4 Strings

Let Σ denote a finite alphabet. We call an element $T \in \Sigma^*$ a *string*. Its length is denoted by $|T|$. The *empty string* is Λ with $|\Lambda| = 0$, such that $\Sigma^+ = \Sigma^* \setminus \{\Lambda\}$. Given an integer j with $1 \leq j \leq |T|$, we access the j-th character of T with $T[j]$. Concatenating a string $T \in \Sigma^*$ k times is abbreviated by T^k. A *bit vector* is a string on the binary alphabet $\{0, 1\}$.

When T is represented by the concatenation of $X, Y, Z \in \Sigma^*$, i.e., $T = XYZ$, then X, Y and Z are called a *prefix*, *substring* and *suffix* of T, respectively; A prefix X, substring Y, or suffix Z is called *proper* if $X \neq T$, $Y \neq T$, or $Z \neq T$, respectively. For i, j with $1 \leq i \leq j \leq |T|$, let $T[i \,..\, j]$ denote the substring of T that begins at position i and ends at position j in T. If $j > i$, then $T[i \,..\, j] = \Lambda$. In particular, the suffix starting at position j of T is called the *j-th suffix* of T, and denoted with $T[j \,..]$. Given a set of text positions $\mathcal{P}$ of T, $\mathit{Suf}(\mathcal{P})$ is the

set of all suffixes whose starting positions belong to $\mathcal{P}$. The *reverse* $T^{\intercal}$ of T is the concatenation $T[n] \cdots T[1] =: T^{\intercal}$.

The *longest common prefix (LCP)* of two strings S and T is the longest string that is a prefix of S and T. The *longest common extension (LCE)* is the LCP of two suffixes of the *same* string. Similarly, the *longest common suffix (LCS)* ending at two positions i and j of the string T is the longest string that is a suffix of $T[..i]$ and $T[..j]$. In particular, we are interested in the lengths of an LCP, LCE, or LCS: The length of the LCP of two strings S and T is the integer $\ell =: \mathsf{lcp}(T, S)$ such that $T[1..\ell] = S[1..\ell]$ and either (a) $T[\ell{+}1] \neq S[\ell{+}1]$ or (b) $\ell = \min(|T|, |S|)$ hold. The length of an LCE is the length of the longest common prefix $T.\mathsf{lce}(i, j) := \mathsf{lcp}(T[i..], T[j..])$ of two suffixes $T[i..]$ and $T[j..]$. The length of the LCS ending at two positions i and j of the string T is the integer $\ell =: T.\mathsf{lcs}(i, j)$ such that $T[i - \ell + 1 .. i] = T[j - \ell + 1 .. j]$ and either (a) $T[i - \ell] \neq T[j - \ell]$ or (b) $\min(i - \ell, j - \ell) = 0$ hold. If the context is clear, we abbreviate $T.\mathsf{lce}$ and $T.\mathsf{lcs}$ to lce and lcs, respectively.

An ordered alphabet Σ induces the *lexicographic order* $\prec$ on the set of strings Σ^*. Given two strings $S, T \in \Sigma^*$, we write $S \prec T$ if S is a proper prefix of T, or $\mathsf{lcp}(S, T) \leq \min(|S|, |T|) - 1$ and $S[1 + \mathsf{lcp}(S, T)] < T[1 + \mathsf{lcp}(S, T)]$.

The *zeroth order empirical entropy* of a string T whose characters are drawn from the alphabet $\Sigma := \{c_1, \ldots, c_\sigma\}$ is $H_0(T) := (1/n) \sum_{j=1}^{\sigma} n_j \lg(n/n_j)$, where n_j is the number of occurrences of the character c_j in T for all integers j with $1 \leq j \leq \sigma$. The *k-th order empirical entropy* of T is $H_k(T) := (1/n) \sum_{S \in \Sigma^k} |T_S| \, H_0(T_S)$, where T_S is the concatenation of each character in T that directly follows an occurrence of the substring $S \in \Sigma^k$ in T. For instance, the substring $S = \texttt{momo}$ of $T = \texttt{momomosumomomo!}$ with alphabet $\{\texttt{m}, \texttt{o}, \texttt{s}, \texttt{u}, \texttt{!}\}$ has four occurrences in T. Concatenating the characters directly following these occurrences yields $T_S = \texttt{msm!}$.

2.5 Regular Structures

Regular structures are analyzed within two main classes: those that are repetitive, and those that are palindromic.[1]

Repetitions. A *period* of a string T is a positive integer $p \leq |T|$ such that $T[i] = T[j]$ for all i and j with $1 \leq i, j \leq |T|$ and $i \equiv j \pmod{p}$. A string can have multiple periods, but only one *smallest period*. The *exponent* $\exp(T)$ of a string T with smallest period p is the (rational) number $|T| / p$. If $\exp(T)$ is at least two, we call the string T *periodic*. A periodic substring $T[i..j]$ is called a *run* if it is *maximal*, i.e., if it can be extended neither (a) to its left ($T[i - 1..j]$) nor (b) to its right ($T[i..j + 1]$) without increasing its smallest

[1] Note that there are also structures that are both repetitive *and* palindromic.

period. Equivalently, the periodic substring $T[i\,..\,j]$ with smallest period p is a run if (a) $i = 1$ or $T[i-1] \neq T[i-1+p]$, and (b) $j = n$ or $T[j+1] \neq T[j+1-p]$.

The currently best known bound on the sum $\mathcal{E}(T)$ of the exponents of all runs in a string T is as follows:

Lemma 2.1 ([21])**.** The sum of the exponents of all runs $\mathcal{E}(T)$ of a string T is less than $3\,|T|$.

It is also known that the number of runs is less than n [21], and that we can find all runs in optimal linear time:

Lemma 2.2 ([160])**.** Given a string T of length n whose characters are drawn from an integer alphabet, we can determine all runs of T in $\mathcal{O}(n)$ time.

The simplest but most analyzed periodic strings are *squares*, i.e., strings of the form SS for $S \in \Sigma^+$. We call S and $|S|$ the *arm* and the *arm length* of the square SS, respectively. The arm length of the square SS is a period of the periodic string SS. A string is called *square-free* if it contains no square.

Palindromes. An (ordinary) *palindrome* S is a string with $S^{\mathsf{T}} = S$. Let T be a string. We call $T[i\,..\,j]$ the *occurrence of a palindrome* in T if $T[i\,..\,j]$ is a palindrome. In such a case, we say that the *center* of this occurrence $T[i\,..\,j]$ is the rational number $(i+j)/2$. An occurrence of a palindrome is called *maximal* if there is no longer palindrome with the same center. Consequently, a maximal palindrome is uniquely defined by its center.

Lemma 2.3. The number of all maximal palindromes in a string of length n is at most $2n - 1$.

Proof. A string of length n has at most $2n - 1$ palindromes with *distinct* centers. □

There is an algorithm computing all maximal palindromes in optimal linear time:

Lemma 2.4 ([12, 182])**.** All maximal palindromes of a string of length n can be computed in $\mathcal{O}(n)$ time in case that two characters can be distinguished in constant time (which is the case for characters of an integer alphabet).

2.6 Support Data Structures

Given a string $T \in \Sigma^*$, a character $\mathsf{c} \in \Sigma$, and an integer j, the *rank* query $T.\mathsf{rank}_{\mathsf{c}}(j)$ counts the occurrences of c in $T[1..j]$, and the *select* query $T.\mathsf{select}_{\mathsf{c}}(j)$ gives the position of the j-th c in T. We stipulate that $\mathsf{select}_1(0) = 0$. If T is a bit vector, there are data structures [49, 135] that use $o(|T|)$ extra bits of space, and can compute rank and select in constant time, respectively. Each

of those data structures can be constructed in time linear in $|T|$. We say that a bit vector has a *rank-support* and a *select-support* if it is endowed by data structures providing constant time access to rank and select, respectively.

Given an integer array A of length n, a *range minimum query (RMQ)* asks for the index of a minimum value in a given range of A. There is a lightweight data structure that can be built on top of A such that it can answer RMQs on A efficiently:

Lemma 2.5 ([86, Thm 3.7])**.** Let $A[1\,..\,n]$ be an integer array, where accessing an element $A[i]$ takes t_A time ($1 \leq i \leq n$). Given a positive constant $\delta \in \mathbb{R}$, there exists a data structure of size δn bits built on top of A that answers RMQs in $\mathcal{O}(t_A/\delta)$ time. It is constructed in $\mathcal{O}(t_A n)$ time with $o(n)$ additional bits of working space.

A predecessor data structure A storing a set S of n elements *sorted* with respect to a linear order $<$ can

- access the i-th smallest element $A[i]$,

and can, for a given element x,

- retrieve the largest index i with $A[i] \leq x$ (*predecessor query*), and
- retrieve the smallest index i with $A[i] \geq x$ (*successor query*).

Let t_{comp} be the time to compare two elements of the set. Given that the elements of S are stored in a sorted integer array A, an access takes constant time, while the other queries are answered in $\mathcal{O}(t_{\text{comp}} \lg n)$ time with a binary search. In a dynamic setting, where it is allowed to add or delete elements of the set, a balanced binary search tree A can answer all queries as well as support the dynamic operations (inserting, changing, and deleting an element) in $\mathcal{O}(t_{\text{comp}} \lg n)$ time.

Lemma 2.6. There is a predecessor data structure storing n elements of a set S taking $\mathcal{O}(n)$ words of space that performs query and update operations in $\mathcal{O}(t_{\text{comp}} \lg n)$ time.

More sophisticated predecessor data structures are described in Sect. 3.2.

2.7 Input Text

For the following chapters, we take a string T of length n as our input, and call it *the text*. All algorithms assume that the characters of T are drawn from an integer alphabet Σ of size $\sigma = |\Sigma| = n^{\mathcal{O}(1)}$. According to Sect. 2.2, each character of T fits into a constant number of memory words. If not stated otherwise, we assume that T is stored in such a way that accessing a character of T costs $\mathcal{O}(1)$

time (e.g., T is stored in RAM using $n \lg \sigma$ bits). When analyzing the memory usage of an algorithm working with T, we neglect the space taken by T. Within Chapter 3, we often assume that T is terminated with a delimiter `$` appearing nowhere else in T, so that no suffix of T is a prefix of another suffix of T. We also assume that `$` is smaller than all other characters of Σ.

Given $\lg \sigma = o(\lg n)$, the RAM model allows us to scan the text in sub-linear time by packing $\lg n / \lg \sigma$ characters into a single word. This technique is called *word-packing*. The idea is to interpret a string T of length n as an integer array with $n / \log_\sigma n$ entries, where each entry takes $\lg n$ bits. Consequently, we can compare $\Omega(\log_\sigma n)$ characters in constant time.

2.8 Effective Alphabet

The restriction to integer alphabets is actually natural when devising string algorithms, since every string T of length n whose characters are drawn from a finite alphabet Σ can be reduced to a string $\tilde{T} \in [1..n]^*$ by sorting T's characters and replacing them with their ranks. We call the new alphabet Σ_T the *effective alphabet* of T. This transformation is natural in the sense that the lexicographic order of all substrings in T is kept ($T[i_1..i_2] \prec T[j_1..j_2] \Leftrightarrow \tilde{T}[i_1..i_2] \prec \tilde{T}[j_1..j_2]$ for $1 \leq i_1 \leq i_2 \leq n$ and $1 \leq j_1 \leq j_2 \leq n$).

The seminal work of Franceschini et al. [97] provides an efficient sorting algorithm for this transformation:

Lemma 2.7 ([97])**.** Given an array of n integers, where each integer fits into a word, we can sort this array in $\mathcal{O}(n)$ time with $\mathcal{O}(\lg n)$ bits of working space.

We present the transformation in two parts. First, we deal with a transformation that maps ranks of Σ_T to characters of Σ:

Lemma 2.8. We can compute an array A with $|\Sigma_T| \lg |\Sigma|$ bits in $\mathcal{O}(n)$ time with $n \lg |\Sigma| + \mathcal{O}(\lg n)$ bits of working space such that $A[\tilde{T}[i]] = T[i]$ for each text position i.

Proof. We copy the text T to an array A of size $n \lg |\Sigma|$ bits, and sort its characters with Lemma 2.7. After sorting A, we remove all adjacent duplicate characters such that $A[i] \in \Sigma$ stores the i-th lexicographically sorted character contained in T, for every integer i from one up to the size of Σ_T. □

Corollary 2.9. There is a data structure computing (a) $\tilde{T}[i]$ with $T[i]$ and (b) $T[i]$ with $\tilde{T}[i]$ in constant time, where $\tilde{T}[1..n] \in \Sigma_T^*$ stores the lexicographic ranks of $T[1..n]$. The data structure takes $|\Sigma_T| \lg |\Sigma| + |\Sigma| \lg |\Sigma_T|$ bits, and can be computed in $\mathcal{O}(n + |\Sigma|)$ time.

Proof. First, we compute the array A of Lemma 2.8. Subsequently, we create an array A^{-1} with $|\Sigma|$ entries and $|\Sigma| \lg |\Sigma_T|$ bits of space. We set $A^{-1}[A[c]] \leftarrow c$ for every rank c with $1 \leq c \leq |\Sigma_T|$. Now we have that $\tilde{T}[i] = A^{-1}[T[i]]$ and $A[\tilde{T}[i]] = T[i]$ for every text position i with $1 \leq i \leq n$. □

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	$
SA	15	14	7	1	11	8	2	12	5	9	3	13	6	10	4
ISA	4	7	11	15	9	13	3	6	10	14	5	8	12	2	1
LCP	0	0	1	5	2	4	4	1	3	4	3	0	2	3	2
BWT	a	b	b	$	b	a	a	a	b	a	a	a	a	a	a

Fig. 2.1: Suffix array, its inverse, the LCP array, and the BWT of $T =$ `aaababaaabaaba$`, as defined in Sect. 2.9.

2.9 Text Data Structures

Assume that $T[n] = \$$. SA and ISA denote the *suffix array* [183] and the *inverse suffix array* of T, respectively. The entry $\mathsf{SA}[i]$ is the starting position of the i-th lexicographically smallest suffix such that $T[\mathsf{SA}[i]\,..] \prec T[\mathsf{SA}[i+1]\,..]$ for all integers i with $1 \leq i \leq n-1$. The *Burrows-Wheeler transform (BWT)* [40] of T is the string BWT with $\mathsf{BWT}[i] = T[n]$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = T[\mathsf{SA}[i]-1]$ otherwise, for every i with $1 \leq i \leq n$. LCP is an array such that $\mathsf{LCP}[i]$ is the length of the LCP of $T[\mathsf{SA}[i]\,..\,n]$ and $T[\mathsf{SA}[i-1]\,..\,n]$ for $i = 2, \ldots, n$. For convenience we stipulate that $\mathsf{LCP}[1] := 0$. The arrays SA and LCP can be constructed in $\mathcal{O}(n)$ time with the algorithms of Ko and Aluru [155] and Kasai et al. [151], respectively. With SA we can construct ISA in $\mathcal{O}(n)$ time by using the fact that SA is a permutation. See Fig. 2.1 for an example of the introduced data structures.

These data structures allow us to answer an LCE query quickly, whose time bound is denoted by t_{LCE}.

Lemma 2.10. Given ISA, LCP and the RMQ data structure of Lemma 2.5 on LCP, we can answer an LCE query on T in $t_{\mathrm{LCE}} = \mathcal{O}(1/\delta)$ time.

Proof. An LCE query $\mathsf{lce}(j_1, j_2)$ with text positions j_1 and j_2 can be answered with an RMQ on the range $[\min(\mathsf{ISA}[j_1], \mathsf{ISA}[j_2]) + 1\,..\,\max(\mathsf{ISA}[j_1], \mathsf{ISA}[j_2])]$ of LCP. □

An LCE data structure is helpful for extending periodic substrings:

Corollary 2.11. Given a string T of length n and an LCE data structure on it, we can return the longest periodic substring with a period p starting at position i in T for integers i and p with $1 \leq i, p \leq n$, in $\mathcal{O}(t_{\mathrm{LCE}})$ time.

Proof. We can compute the LCE of $T[i\,..]$ and $T[i+p\,..]$ in $\mathcal{O}(t_{\mathrm{LCE}})$ time. If this prefix is $T[i+p\,..\,\ell]$, then $T[i\,..\,\ell]$ is the longest periodic substring with a period p starting at position i. □

There are two easy corollaries of Lemma 2.10 that are helpful for LCS queries or for finding palindromic strings:

Corollary 2.12. Given a string T of length n, there is a data structure that answers the queries $\mathsf{lcp}(T[i\,..], T[j\,..])$ and $\mathsf{lcs}(T[i\,..], T[j\,..])$ in constant time. The data structure can be built in $\mathcal{O}(n)$ time, taking $\mathcal{O}(n)$ words of space.

Proof. We build the data structures of Lemma 2.10 on T and its reverse. Then $\mathsf{lcs}(T[i\,..], T[j\,..]) = \mathsf{lcp}(T^{\intercal}[n+1-i\,..], T^{\intercal}[n+1-j\,..])$. □

We write $\mathsf{LCE}_T^{\leftrightarrow} = \mathsf{LCE}^{\leftrightarrow}$ for a data structure supporting the queries of Cor. 2.12 with t_{LCE} time.

Corollary 2.13. Given a string T of length n, there is a data structure that can compute the longest prefix of $T[i\,..]$ that is a suffix of $T[1\,..\,j]$. The data structure can be built in $\mathcal{O}(n)$ time, taking $\mathcal{O}(n)$ words of space.

Proof. We compute the data structures of Lemma 2.10 on the string $S \coloneqq T[1]\dots T[n]T[n-1]\dots T[1]$.[2] Then the longest prefix of $T[i\,..]$ that is a suffix of $T[1\,..\,j]$ is given by $\mathsf{lcp}(S[i\,..], S[2n-j\,..])$. □

[2] Actually, the data structures of Lemma 2.10 require that S ends with a unique delimiter that is smaller than all its characters (like $\$ = T[n]$). For that, we simply append a unique delimiter to S that is lexicographically smaller than all characters of $\Sigma \cup \{\$\}$.

Chapter

3

Lempel-Ziv Factorizations

> *Any string of symbols that can be given an abbreviated representation is called* algorithmically compressible. *On this view, we recognize science to be a search for algorithmic compressions.*
>
> — John D. Barrow [25]

A preliminary task of most text compressors is to factorize a text into substrings. This task is also called *text factorization*. Probably the most famous such text factorizations are the Lempel-Ziv-77 (LZ77) [246] and Lempel-Ziv-78 (LZ78) [247] factorizations. Their area of application is widespread (cf. [214]), e.g., in the command-line compression tools `gzip` (for LZ77) or `compress` (for LZ78), in image file formats like `png` (using LZ77) or `gif` (using LZ78). The Lempel-Ziv (LZ) factorizations have been found valuable for string dictionaries [16], for text indexes [13, 78, 101, 102, 142, 143, 192], for detecting various kinds of regularities in strings (like [57, 71, 121, 156, 160, 161, 181] or Sect. 3.5), and for analyzing strings [61, 175, 176]. While the compression rates of LZ77-based compressors are practically superior to those based on LZ78 in most cases, the biggest advantage of LZ78 over LZ77 is that LZ78 allows for an easy construction *within compressed space* and in *near-linear time*, which is not possible (to date) for LZ77.

Although the family of LZ factorizations is well-studied, there are still improvements being made in computing them with respect to space and time. Computing the factorizations efficiently is of practical interest, since main memory sizes of ordinary computers do not scale as fast as the sizes of commonly maintained datasets. Both huge mainframes with massive datasets and tiny embedded systems are valid examples for which a simple compressor may end up depleting all available RAM. Additionally, compression algorithms with low memory footprints are good candidates for compressing transient data kept in memory. For instance, the *zram* module of modern Linux kernels [211] compresses blocks of the main memory to prevent the system from running out of working memory. Compressing RAM is sometimes preferable to storing transient data on secondary storage (e.g., in a swap file), as the latter poses a more severe performance loss. Another example is transferring websites as compressed data by hosting servers [204]. A server may cache generated web pages in a compressed form in RAM for performance benefits. To solve this

problem, one has to think either about external memory compression algorithms or about how to slim down memory consumption during computation in RAM. In this chapter we present algorithms belonging to the latter approach.

3.1 Our Contribution

In Sect. 3.4 and Sect. 3.7 we present LZ77 and LZ78 factorization algorithms using two suffix tree representations whose construction algorithms need only limited working space. The used representations differ in the fact that one is favorable for small alphabets, while the other is independent of the alphabet size. Powered by these two suffix tree representations, we present algorithms computing the LZ77 and the LZ78 factorization

- in $\mathcal{O}(n)$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space (Cor. 3.14 for LZ77 and Cor. 3.41 for LZ78), or
- in $\mathcal{O}(n/\epsilon)$ time with $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of working space (Cor. 3.17 for LZ77 and Cor. 3.48 for LZ78), where $\epsilon \in \mathbb{R}$ is a trade-off parameter that can be selected within the domain $0 < \epsilon \leq 1$.

In the following, we treat ϵ as a real-valued constant such that $\mathcal{O}(n/\epsilon)$ becomes $\mathcal{O}(n)$ (cf. Figs. 3.1 to 3.3). Up to $\lg \sigma = o(\lg n)$, the former algorithms with $\mathcal{O}(n \lg \sigma) = o(n \lg n)$ bits are favorable to the latter ones. The working space bounds of the $\mathcal{O}(n \lg \sigma)$-bits algorithms are due to the suffix tree: Given that we have constant time access to the Ψ function (cf. Sect. 3.3) and to the tree navigational operations of the suffix tree (cf. Sect. 3.3.3), we can compute both factorizations in linear time using $z \lg n + \mathcal{O}(n)$ additional bits of working space (Thms. 3.13 and 3.40), where z is the number of factors in the LZ78 factorization.

The time and space bounds of our LZ77 factorization algorithms hold for two popular LZ77 schemes:

- the original version of Ziv and Lempel [246], which we call *classic-LZ77*, and
- the variant of Storer and Szymanski [224], which we just call *LZ77*, since most research articles on the LZ77 factorization refer to this specific variant with that name.

We also study the non-overlapping LZ77 factorization (Thm. 3.35) and the reversed LZ77 factorization (Thm. 3.37), for which we can adapt our LZ77 factorization algorithms with a small penalty with respect to time and space.

As an application of the LZ77 factorization algorithms, we show how to compute the longest previous factor (LPF) table and the number of all distinct squares. The former is an array of the same length as the text whose i-th

entry stores the length of the longest substring that starts before i and is a prefix of the suffix starting at i. We show that we can compute the LPF table in a $2n + o(n)$-bits representation (Cor. 3.21) with either $\mathcal{O}(n \lg^{\epsilon}_{\sigma} n)$ time and $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits of working space, or $\mathcal{O}(\epsilon^{-1} n)$ time and $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of working space (Lemma 3.22). Subsequently, the LPF table leads us to the set of all distinct squares in Sect. 3.5, which has applications by itself.

In the last section of this chapter, we present a practical Las Vegas algorithm computing the LZ78 factorization and one of its variants, the Lempel-Ziv-Welch (LZW) factorization, in $\mathcal{O}(n/\epsilon)$ time with $\mathcal{O}(z \lg(z\sigma))$ bits of space (Thm. 3.52). The algorithm uses a trie data structure based on compact hashing [50]. We also present a trie data structure based on Karp-Rabin fingerprints [149]. Together with other trie data structures, we conduct a thorough evaluation of the LZ78 and LZW computation. It turns out that the trie based on compact hashing has the lowest memory usage, while the trie based on fingerprinting is the quickest. The takeaway message of Sect. 3.8 is that one can beat well-tuned out-of-the-box trie data structures like Judy[1], m-Bonsai [206], or the Cedar-trie [245] with a careful choice of the LZ trie representation.

3.2 Related Work

While there are naïve algorithms with quadratic running time (for both LZ77 and LZ78), algorithms with limited space requirements running in (nearly) linear time have only emerged in recent years (see Figs. 3.1 to 3.3). In what follows, we give an overview of the most recent algorithms computing the LZ77 and LZ78 factorization.

First, let us look at the LZ77 factorization algorithms that run in linear time. There, Goto and Bannai [115] showed an algorithm using $3n \lg n$ bits. This bound was very soon lowered to $2n \lg n$ by Kärkkäinen et al. [148]. With similar techniques, Goto and Bannai [116] proposed later a solution with $n \lg n + \mathcal{O}(\sigma \lg n)$ bits, which is compelling if the alphabet size σ is small. The common idea of the above articles is the usage of previous- and/or next-smaller-value queries [83]. While Kärkkäinen et al. [148] need to hold the suffix array and the next-smaller-value array in two arrays, Goto and Bannai [116] can cope with a single $n \lg n$ bits array and an auxiliary array of size $\mathcal{O}(\sigma \lg n)$ bits. This auxiliary array is used to store the bucket boundaries needed by the suffix array construction algorithm of Nong [199]. The bucket boundaries could also be represented by a dynamic bit vector [195] of length n, yielding $n + o(n)$ additional bits (instead of the $\mathcal{O}(\sigma \lg n)$ bits used by the auxiliary array) of working space and $\mathcal{O}(n \lg n / \lg \lg n)$ time.

Another algorithm close to linear time was devised by Kempa and Puglisi [152]. They show a practical variant with $(1 + \epsilon)n \lg n + n + \mathcal{O}(\sigma \lg n)$ bits of work-

[1] http://judy.sourceforge.net

LZ77 factorization algorithms		
Time	Working Space	Ref.
$\mathcal{O}(n)$	$3n \lg n$	[115]
$\mathcal{O}(n)$	$2n \lg n$	[148]
$\mathcal{O}(n)$	$n \lg n + \mathcal{O}(\sigma \lg n)$	[116]
$\mathcal{O}(n)$	$(1+\epsilon)n \lg n + \mathcal{O}(n)$	Cor. 3.17
$\mathcal{O}(n)$	$\mathcal{O}(n \lg \sigma)$	Cor. 3.14
$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(n \lg n)$	[213]
$\mathcal{O}\left(n \frac{\lg n}{\lg \lg n}\right)$	$n \lg n + n + o(n)$	[116, 195]
$\mathcal{O}(n \lg \sigma)$	$(1+\epsilon)n \lg n + n + \mathcal{O}(\sigma \lg n)$	[152]
$\mathcal{O}(n \log_\sigma n \lg \lg \sigma)$	$\mathcal{O}(n \lg \sigma)$	[147]
$\mathcal{O}(n \lg \lg \sigma)$	$\mathcal{O}(n \lg \sigma)$	[27]

LZ78 factorization algorithms		
Time	Working Space	Ref.
$\mathcal{O}(n)$	$\mathcal{O}(n \lg n)$	[191]
$\mathcal{O}(n)$	$(1+\epsilon)n \lg n + \mathcal{O}(n)$	Cor. 3.48
$\mathcal{O}(n)$	$\mathcal{O}(n \lg \sigma)$	Cor. 3.41
$\mathcal{O}(n \lg \sigma)$	$\mathcal{O}(z \lg z)$	folklore
$\mathcal{O}(n(\lg \sigma + \lg \lg n))$	$z \lg n + z \lg \sigma + \mathcal{O}(z)$	[13]
$\mathcal{O}\left(\frac{n(\lg \sigma + \lg \lg n)}{\lg \lg n}\right)$	$z \lg n + z \lg \sigma + \mathcal{O}(z)$	[14]
$\mathcal{O}\left(\frac{n}{\log_\sigma n} \frac{\lg^2 \lg n}{\lg \lg \lg n}\right)$	$\mathcal{O}\left(n \frac{\lg \sigma + \lg \log_\sigma n}{\log_\sigma n}\right)$	[137]
$\mathcal{O}\left(n + z \frac{\lg^2 \lg \sigma}{\lg \lg \lg \sigma}\right)$	$\mathcal{O}(z \lg z)$	[85]

Fig. 3.1: Recent approaches in LZ77 (*top*) and LZ78 (*bottom*) factorization computation. The working spaces are measured in bits (additional $\mathcal{O}(\lg n)$ bits omitted). The horizontal line in each table separates algorithms running in linear time (*above*) from algorithms running in near-linear time (*below*).

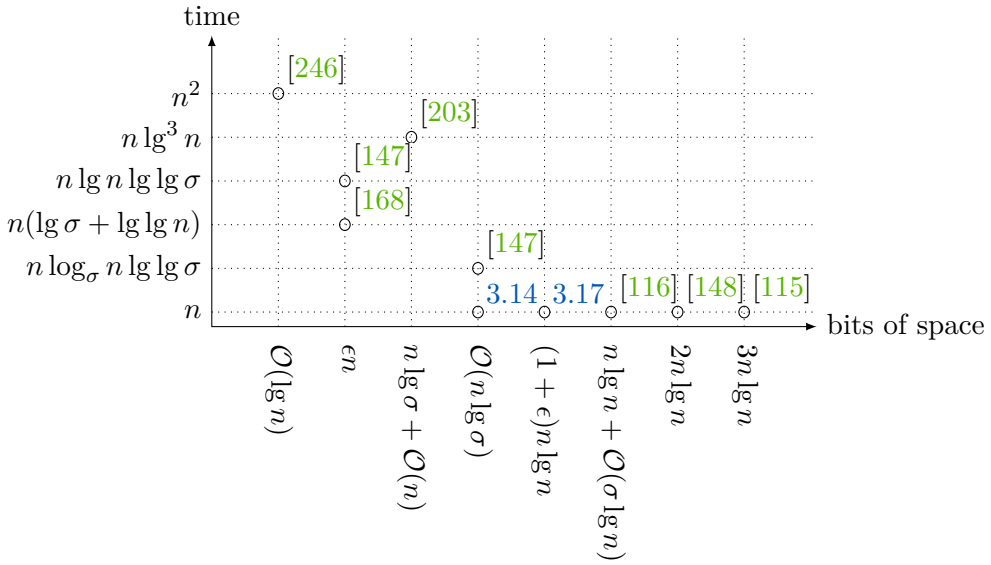


Fig. 3.2: Space/Time plot of LZ77 algorithms. The time is measured in $\mathcal{O}(\cdot)$. The numbers 3.14 and 3.17 refer to Cors. 3.14 and 3.17 of this chapter, respectively.

Time	Working Space	Ref.
$\mathcal{O}(n \lg^3 n)$	$n \lg \sigma + \mathcal{O}(n)$	[203]
$\mathcal{O}(n \lg n \lg \lg \sigma)$	ϵn	[147]
$\mathcal{O}(n(\lg \sigma + \lg \lg n))$	ϵn	[168]

Fig. 3.3: LZ77 algorithms with small working spaces, which are measured in bits.

ing space and $\mathcal{O}(n \lg \sigma / \epsilon^2)$ time. There is also a trade-off algorithm given by Kärkkäinen et al. [147], using $\mathcal{O}((n \lg n)/d)$ bits of working space and $\mathcal{O}(dn \lg \lg \sigma)$ time, where the $\mathcal{O}(\lg \lg \sigma)$ time factor is due to a wavelet tree [23]. Setting $d \leftarrow \log_\sigma n$ we obtain $\mathcal{O}(n \lg \sigma)$ bits of working space and $\mathcal{O}(n \log_\sigma n \lg \lg \sigma)$ time.

The last group of LZ77 factorization algorithms that is notable to mention consists of those algorithms that use the suffix tree. The first such algorithm is by Rodeh et al. [213]. It uses $\mathcal{O}(n \lg n)$ bits of space and runs in linear time for constant alphabets. A more recent approach was presented by Belazzougui and Puglisi [27, Sect. 4], whose algorithm runs in $\mathcal{O}(n)$ time with $\mathcal{O}(n \lg \sigma)$ additional bits of space on top of the suffix tree. With the compressed suffix tree representation of Munro et al. [190], their algorithm has the same time and space bounds as our LZ77 factorization algorithms (Cor. 3.14) running in $\mathcal{O}(n)$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space.

The space bounds can become more attractive if even slower execution times are affordable (cf. Fig. 3.3). We start with an approach of Okanohara and Sadakane [203], which runs in $\mathcal{O}(n \lg^3 n)$ time using $n \lg \sigma + \mathcal{O}(n)$ bits. The aforementioned algorithm by Kärkkäinen et al. [147] works with $\mathcal{O}(\epsilon n)$ bits and $\mathcal{O}(n \lg n \lg \lg \sigma)$ time by setting $d \leftarrow \epsilon \lg n$ for a constant ϵ with $0 < \epsilon$.

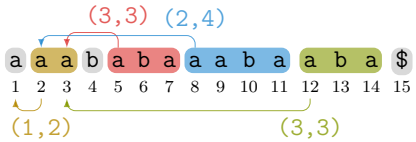
Within the same space bound, Kosolobov [168] presented an algorithm running in $\mathcal{O}(n(\lg\sigma + \lg\lg n))$ time.

The LZ78 factorization is done with different techniques. An LZ78 factorization of size z can be stored in two arrays with $z\lg\sigma$ and $z\lg z$ bits to represent the character (belonging to an alphabet of size σ) and the referred index, respectively, of each factor. This space bound has not yet been achieved by any efficient dynamic trie data structure. The classic approach is to build a dynamic trie maintaining the factors during the computation. Storing z factors in a simple pointer-based trie data structure takes $\mathcal{O}(z\lg z) = \mathcal{O}(n\lg\sigma)$ bits (see Lemma 3.4 why this holds) and $\mathcal{O}(n\lg\sigma)$ time, if the children of a node are maintained in a sorted list. Other ways to improve the LZ78 computation are by using sophisticated trie implementations [85, 137], or by superimposing the suffix tree with the suffix trie [191].

Following the former approach, Jansson et al. [137] proposed a compressed dynamic trie based on word packing, and showed an application computing the LZ78 factorization with $\mathcal{O}(n(\lg\sigma + \lg\log_\sigma n)/\log_\sigma n)$ bits of working space and $\mathcal{O}(n\lg^2\lg n/(\log_\sigma n\lg\lg\lg n))$ time. When $\lg\sigma = o(\lg n\lg\lg\lg n/\lg^2\lg n)$, their algorithm runs even in sub-linear time, but in the worst case it is super-linear. More sophisticated trie implementations [85] improve this to $\mathcal{O}(n + z\lg^2\lg\sigma/\lg\lg\lg\sigma)$ time, while using $\mathcal{O}(z\lg z)$ bits of space like the naïve trie implementation. The time bound $\mathcal{O}(n + z\lg^2\lg\sigma/\lg\lg\lg\sigma)$ becomes linear when $\lg\sigma = o(\lg n\lg\lg\lg n/\lg^2\lg n)$. Versions with succinct data structures are shown by Arroyuelo and Navarro [13, Lemma 8] and Arroyuelo et al. [14] using $2z\lg z + z\lg\sigma + \mathcal{O}(z)$ bits of working space and either $\mathcal{O}(n(\lg\sigma + \lg\lg n))$ or $\mathcal{O}((n\lg\sigma)/\lg\lg n)$ time (for $\sigma = \omega(\lg^{\mathcal{O}(1)} n)$ and $\sigma = \mathcal{O}(n)$), respectively. All these tries are favorable for small alphabet sizes. If the alphabet size σ becomes large, the upper bounds on the times become unattractive.

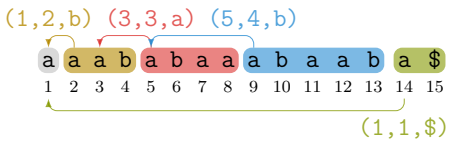
Our theoretical LZ78 factorization algorithms follow the latter approach that superimposes the suffix tree with the suffix trie. There, Nakashima et al. [191] recently proposed a linear time algorithm. Although their algorithm works with $\mathcal{O}(n\lg n)$ bits of space, they did not care about the constant factor, and the use of the (complicated) dynamic marked ancestor queries [6] seems to prevent them from achieving a small constant factor.

On the practical side, we are only aware of the approaches of Arroyuelo et al. [15] and Navarro [193]. The approach of Arroyuelo et al. [15] leverages compact hashing to compute the LZ78 factorization with an alternative coding within $\mathcal{O}(z\lg\sigma)$ bits of working space in $\mathcal{O}(n)$ randomized time. Unfortunately, none of them provides a systematic study of practical LZ78 computation algorithms. A study related to what we present here was conducted by Kanda et al. [140], but with focus on dynamic tries storing arbitrary strings (i.e., the edge labels of their tries are strings, not restricted to single characters).



Coding: a(1,2)b(3,3)(2,4)(3,3)$

(a) LZ77 Factorization



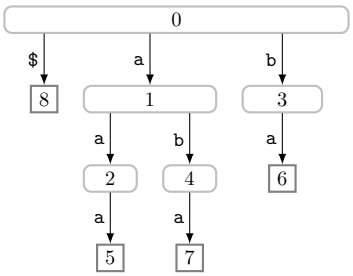
Coding: a(1,2,b)(3,3,a)(5,4,b)(1,1,$)

(b) Classic-LZ77 Factorization

Fig. 3.4: Two kinds of LZ77 factorizations of the text `aaababaaabaaba$`. The coding represents a fresh factor by a single character, and a referencing factor by a tuple with two or three entries. For LZ77 (a), this tuple consists of the referred position and the number of characters to copy. For classic-LZ77 (b), the tuple additionally contains the character at the end of the factor.

i	1	2	3	4
Factor	a	aa	b	ab
Coding	(0,a)	(1,a)	(0,b)	(1,b)
i	5	6	7	8
Factor	aaa	ba	aba	$
Coding	(2,a)	(3,a)	(4,a)	(0,$)

(a) LZ78 Factorization



(b) LZ Trie

Fig. 3.5: The LZ78 factorizations of the text of Fig. 3.4. The coding in (a) is a list of pairs. Each pair consists of the referred index and the character at the end of the respective factor. The LZ trie (b) is another representation of the LZ78 factorization.

3.3 Preliminaries

We first define the LZ77 and LZ78 factorizations. Subsequently, we present our two suffix tree representations with which we compute the LZ77 and LZ78 factorizations in Sect. 3.4 and Sect. 3.7, respectively. Finally, we stipulate a common framework of our factorization algorithms.

3.3.1 Factorizations

A *factorization* of a string T is a sequence of non-empty substrings $F_1, \ldots, F_z$ of T such that the concatenations of the substrings $F_1 \cdots F_z$ yields T. Each substring of the factorization is called a *factor*.

Definition 3.1. A factorization $F_1 \cdots F_z = T$ is called the *LZ77 factorization* of T if $F_x = \text{argmax}_{S \in \mathcal{S}_j(T) \cup \Sigma} |S|$ for all x with $1 \leq x \leq z$ and $j := |F_1 \cdots F_{x-1}| + 1$,

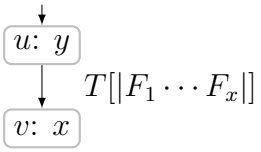


Fig. 3.6: Two connected nodes of an LZ trie. The nodes u and v correspond to the y-th and x-th LZ78 factor, respectively. Since v is the child of u, the x-th factor refers to the index y, and consequently $F_x = F_y T[|F_1 \cdots F_x|]$.

where $\mathcal{S}_j(T)$ is the set of substrings of T that start *strictly* before j.

In other words, the LZ77 factor F_x is either (a) the longest prefix of $F_x \cdots F_z$ that occurs at least twice in $F_1 \cdots F_x$, or (b) equal to $T[1 + |F_1 \cdots F_{x-1}|]$ if such a prefix does not exist. This factorization is also called *s-factorization* [56] or *Lempel-Ziv-Storer-Szymanski (LZSS)* factorization [224]. An example of the LZ77 factorization of the text `aaababaaabaaba$` is given in Fig. 3.4a.

The original factorization by Ziv and Lempel [246] is actually a variant of the here introduced LZ77 factorization. We call this variant the *classic-LZ77* factorization. The difference is that each factor of the classic-LZ77 factorization introduces an additional character at its end:

Definition 3.2. A factorization $F_1 \cdots F_z = T$ is called the *classic-LZ77 factorization* of T if F_x is the shortest prefix of $F_x \cdots F_z$ that occurs exactly once in $F_1 \cdots F_x$.

We refer to Fig. 3.4b for an example of the classic-LZ77 factorization.

Definition 3.3. A factorization $F_1 \cdots F_z = T$ is called the *LZ78 factorization* of T if $F_x = F'_x c$ with $F'_x = \text{argmax}_{S \in \{F_y | y<x\} \cup \{\Lambda\}} |S|$ and $c \in \Sigma$ for all $1 \leq x \leq z$.

To put it in different words, the LZ78 factor F_x is the longest prefix of $T[1 + |F_1 \cdots F_{x-1}| \,..]$ that is equal to $F_y c$ for an index y with $0 \leq y \leq x-1$ and a character $c \in \Sigma$ (we stipulate that $F_0 \coloneqq \Lambda$). See Fig. 3.5 for an example of the LZ78 factorization.

Although the above defined factorizations do not share the same value of z in general, there is a well-known upper bound of z for all these factorizations:

Lemma 3.4 ([247])**.** Given a text of length n on an alphabet of size σ, the LZ77, LZ77-classic, or LZ78 factorization divides the text into z factors with $z \leq \mathcal{O}(n/\log_\sigma n)$. We obtain that $\mathcal{O}(z \lg z) = \mathcal{O}(z \lg n) = \mathcal{O}(n \lg \sigma)$.

We call a text position j with $1 \leq j \leq n$ the *starting position* of the factor F_x with $1 \leq x \leq z$ if F_x starts at position j. A factor F_x may refer to either

(LZ77) a previous text position j (called F_x's *referred position*), or

(LZ78) to a previous factor F_y (called F_x's *referred factor*—in this case y is also called the *referred index* of F_x).

If there is no suitable reference found for a given factor F_x with starting position j, then F_x consists of just the single letter $T[j]$. We call such a factor a *fresh factor*. Fresh LZ78 factors refer to $F_0 \coloneqq \Lambda$. The other factors are called *referencing factors*; let z_{R} denote the number of referencing factors.

A natural representation of the LZ78 factors is a trie, the so-called *LZ trie*. Except for the root, each node in the LZ trie represents a factor and is labeled with its factor index (see Fig. 3.5b). The trie has the following properties: If the x-th factor refers to the y-th factor, then there is a node u having a child v such that u and v have the unique labels y and x, respectively. The edge (u, v) is labeled with the last character of the x-th factor. Figure 3.6 depicts such an edge. A node with label x is the child of the root if and only if the x-th factor is a fresh factor.

3.3.2 Suffix Trees

The *suffix trie* of T is the trie of all suffixes of T. Each suffix trie edge e is associated with a character of T called the *edge label* of e. The *suffix tree* of T is the tree obtained by compacting the suffix trie of T, i.e., by contracting each unary path to a single edge e, which is associated with the concatenation of the labels of the edges on the contracted path; the string yielded by this concatenation is called the *edge label* of e. Consequently, the edge label of a suffix tree edge is a non-empty *substring* of T. We define the function $\mathsf{edge\_length}(e)$ returning, for each edge e, the length of e's label. The *string label* of a node v is defined as the concatenation of all edge labels on the path from the root to v; the *string depth* of a node is the length of its string label. By construction, the suffix tree has n leaves and at most $n-1$ internal nodes. The leaf corresponding to the i-th suffix with $1 \leq i \leq n$ is associated with *suffix number* i (see the underlined numbers in Fig. 3.7). We write $\mathsf{sufnum}(\lambda)$ to denote the suffix number of a leaf λ. Reading the suffix numbers in preorder gives the suffix array. This means that the preorder of the leaves is an enumeration of the leaves according to the lexicographic order of their respective suffixes. However, we sometimes have neither the function sufnum nor SA available. In most cases, we do not represent the suffix array as a plain array such that the access time to SA, which we denote by t_{SA}, is not constant in general.

A pointer-based suffix tree of a text T with length n takes $\mathcal{O}(n \lg n)$ bits. Farach-Colton et al. [75] presented an algorithm constructing the pointer-based suffix tree of T in $\mathcal{O}(n)$ time. This representation supports the operations listed in the following section.

3.3.3 Operations on the Suffix Tree

The algorithms introduced in Sects. 3.4 to 3.7 operate on suffix trees and need the following navigational methods (v is a suffix tree node, λ and λ' are suffix

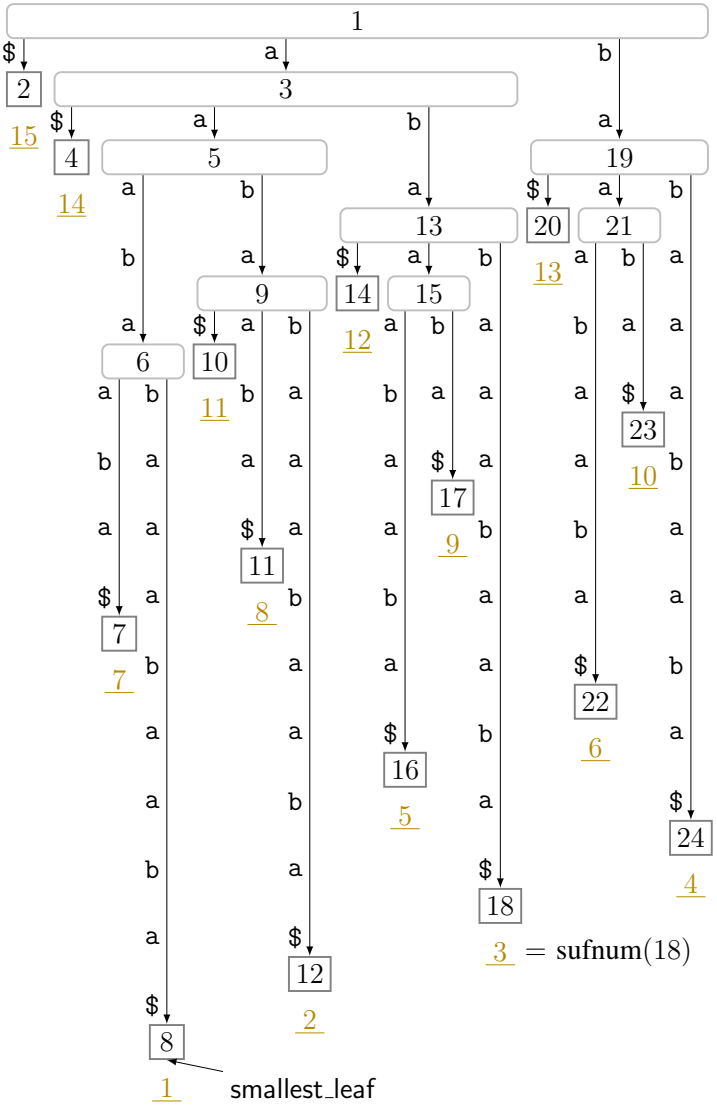


Fig. 3.7: The suffix tree of T = `aaababaaabaaba$`. The nodes are labeled by their preorder numbers, which are induced by a preorder traversal of the tree. The suffix number of each leaf λ is the underlined number drawn in dark yellow below λ.

tree leaves; v, λ and λ' are represented by their preorder numbers):

- **parent**(v) returns the parent of the node v,
- **depth**(v) returns the tree depth of the node v,
- **level_anc**(λ, d) returns the ancestor at depth d of the leaf λ,
- **leaf_select**(i) returns the i-th leaf (in lexicographic order) for an integer i with $1 \leq i \leq n$,
- **lca**(λ, λ') returns the lowest common ancestor (LCA) of the leaves λ and λ',
- **lmost_leaf**(v) and **rmost_leaf**(v) return the leftmost and rightmost leaf below the node v, respectively,

- leaf_rank(λ) returns the number of preceding leaves (in lexicographic order) of the leaf λ incremented by one (such that leaf_rank(lmost_leaf(root)) $= 1$ and leaf_rank(rmost_leaf(root)) $= n$, where root is the root),
- child_rank(v) returns the number of preceding siblings of the node v,
- v.child(i) returns the i-th child of the node v (we only need $i = 1, 2$), and
- next_sibling(v) returns the next sibling of the node v.

Note that leaf_rank and sufnum are related, but different (we require access only to the former operation). The difference is that, given a leaf λ representing the i-th suffix, sufnum(λ) $= i$, whereas leaf_rank(λ) $=$ ISA$[i]$, i.e., sufnum(λ) $=$ SA[leaf_rank(λ)].

Additionally, we address the leaf with suffix number 1 with smallest_leaf, and want access to the following specific methods:

- head(λ) returns the first character of the suffix whose starting position is sufnum(λ),
- next_leaf(λ) returns the leaf with suffix number sufnum(λ) $+ 1$ or the leaf with label 1 (i.e., smallest_leaf) if sufnum(λ) $= n$, and
- str_depth(v) returns the string depth of the node v, given that v is an *internal* node.

To attain the promised upper bound of either $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits or $\mathcal{O}(n \lg \sigma)$ bits of working space, we focus on representations of the suffix tree that are especially trimmed on a small memory footprint during their construction. Unfortunately, most of the suffix tree representation need more space during their construction, e.g., the representation of Russo et al. [215] can be stored in $nH_k(T) + o(n \lg \sigma)$ bits, but it needs $\Omega(n \lg \sigma)$ bits of working space during its computation.

In the following, we present two suffix tree representations with a memory-efficient construction, and show how the above methods can be computed with each representation. The first one, called compressed suffix tree, uses $\mathcal{O}(n \lg \sigma)$ bits of total space that is favorable for small alphabet sizes. In case of a large alphabet, we choose an alphabet-independent solution, which we call succinct suffix tree.[2] It uses $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of total space; its memory footprint is independent of the size of the alphabet. The compressed suffix tree uses less memory than the succinct suffix tree for $\lg \sigma = o(\lg n)$, but does not have any advantages over even the pointer-based representation when $\sigma = \Omega(n)$.

[2] The name succinct does not mean that the suffix tree is stored in succinct space, but rather that it consists of an assembly of data structures, where most of them are stored succinctly.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	a	a	a	b	a	b	a	a	a	b	a	a	b	a	$
SA	15	14	7	1	11	8	2	12	5	9	3	13	6	10	4
ISA	4	7	11	15	9	13	3	6	10	14	5	8	12	2	1
Ψ	4	1	6	7	8	10	11	12	13	14	15	2	3	5	9
LCP	0	0	1	5	2	4	4	1	3	4	3	0	2	3	2
PLCP	5	4	3	2	3	2	1	4	4	3	2	1	0	0	0
LPF	0	2	1	0	3	2	5	4	4	3	4	3	2	1	0

Fig. 3.8: Figure 2.1 augmented with the Ψ-function, PLCP (Sect. 3.3.3.2) and LPF (Sect. 3.4.4) of $T =$ aaababaaabaaba$. Reading the suffix numbers of the leaves of the suffix tree in Fig. 3.7 from left to right gives SA.

3.3.3.1 Compressed Suffix Tree

The *compressed suffix tree (CST)* consists of two components. The first data structure is the Ψ-function [118] (see also Fig. 3.8). It is defined by

$$\Psi[i] := \begin{cases} \mathsf{ISA}[1] & \text{if } \mathsf{SA}[i] = n, \\ \mathsf{ISA}[\mathsf{SA}[i]+1] & \text{otherwise.} \end{cases}$$

It can be stored in $\mathcal{O}(n \lg \sigma)$ bits while allowing constant access time [124]. The second one is a balanced parentheses (BP) representation [135] of the tree topology [217] taking $4n + o(n)$ bits. Munro et al. [190] show the following result:

Theorem 3.5 ([190])**.** The compressed suffix tree takes $\mathcal{O}(n \lg \sigma)$ bits of space. We can construct it with $\mathcal{O}(n \lg \sigma)$ bits of working space in $\mathcal{O}(n)$ time.

The compressed suffix tree supports all operations described in Sect. 3.3.3: First, the BP sequence supports the navigational methods in constant time [136, 197]. Second, we have access to smallest_leaf because leaf_rank(smallest_leaf) $=$ $\Psi[1] =$ ISA$[1]$. To see that, we remember that \$ is the smallest character in T appearing exactly once at $T[n]$; hence SA$[1] = n$. Finally, we can implement the remaining methods efficiently as follows:

head(λ) Given that Σ is the *effective* alphabet of T, the root of the suffix tree has σ children, each corresponding to a different character. The orders of the children of the root and of the characters of Σ are the same. Hence, child_rank(level_anc$(\lambda, 1)$) $=$ head(λ) holds, and the left hand side can be computed in constant time.

If Σ is not effective, we build the array A of Lemma 2.8. Having A, we compute head(λ) with A[child_rank(level_anc$(\lambda, 1)$)] in constant time.

Algorithm 1: Computing the string depth of a suffix tree node.

```
   input : suffix tree node v
1  function str_depth
2      if v is an internal node then
3          λ ← lmost_leaf(v.child(1))
4          λ' ← lmost_leaf(v.child(2))          ▷ exists since v is an internal node
5          m ← 0
6          while head(λ) = head(λ') do
7              λ ← next_leaf(λ)
8              λ' ← next_leaf(λ')
9              incr m
10         return m
11     else if v is a leaf then                 ▷ works only if sufnum(v) is available
12         return n + 1 − sufnum(v)
```

next_leaf(λ) We can compute **next_leaf**(λ) = **leaf_select**(Ψ[**leaf_rank**(λ)]) in constant time.

str_depth(v) We use the Ψ-function and **head** to compute the string depth of v in time proportional to the string depth (see Algo. 1): First, we take two different children of v (they exist since v is an internal node), and choose an arbitrary leaf in the subtree of each child. By doing so, we have selected two leaves representing two suffixes whose LCP is the string label of the LCA of both leaves. Our task is to compute the length of this prefix. To this end, we match the first characters of both suffixes by **head**. If they match, we use Ψ to move to the next pair of suffixes[3], and apply **head** again. Informally, applying Ψ strips the first character of both suffixes (like taking a suffix link, i.e., an edge from a node whose string label is cS to a node whose string label is S, with $c \in \Sigma, S \in \Sigma^*$). We repeat this as long as the first characters of both suffixes match. On a mismatch of the first characters, each character finally represents an edge from v to a different child, i.e., we have found the string depth of v that is equal to the number of matched characters.

Usually, we do not need to compute **sufnum**(λ) = SA[**leaf_rank**(λ)]. Since the compressed suffix tree does not provide access to SA, such a call would cost us $\mathcal{O}(n)$ time by sequentially scanning all leaves in lexicographic order. However, in some scenarios we need access to SA (Sects. 3.4.4 and 3.6). For that, we can augment the CST with the following data structure that gives access to SA:

[3] Meaning that we select those two leaves that have the respectively next larger suffix numbers.

Lemma 3.6 (Grossi and Vitter [118, Sect. 3.2])**.** There is a data structure using $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits that can access SA in $\mathcal{O}(\lg_\sigma^\epsilon n)$ time, where ϵ is a constant with $0 < \epsilon \leq 1$.

Belazzougui et al. [28] present an overview for other approaches supporting access to SA.

3.3.3.2 Succinct Suffix Tree

Remembering Chapter 2, the arrays SA and ISA already need $2n \lg n$ bits of space, which is too much for our aimed space bounds of $(1+\epsilon)n \lg n$ bits. We cannot store both SA and ISA as plain arrays. Instead, we make use of the following data structure:

Lemma 3.7 ([194, Sect. 5.2])**.** Given a permutation A of the integers $[1 \,..\, n]$, there is a data structure that provides access to A's inverse in $\mathcal{O}(1/\epsilon)$ time. The data structure uses $\epsilon n \lg n + n$ bits. It can be constructed with additional n bits in $\mathcal{O}(n)$ time.

Subsequently, we examine the space of LCP, which also needs $n \lg n$ bits as a plain array. Sadakane [217] presented a representation of LCP taking $2n + o(n)$ bits. His idea is to store the *permuted longest-common-prefix array* PLCP, defined as $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$, in a bit vector in the following way (also described in [81]): Since $\mathsf{PLCP}[1] + 1, \mathsf{PLCP}[2] + 2, \ldots, \mathsf{PLCP}[n] + n$ is a non-decreasing sequence with $1 \leq \mathsf{PLCP}[1] + 1 \leq \mathsf{PLCP}[n] + n = n$ ($\mathsf{PLCP}[i] \leq n - i$ since the terminal $\$$ is a unique character in T) the values $I[1] := \mathsf{PLCP}[1]$ and $I[i] := \mathsf{PLCP}[i] - \mathsf{PLCP}[i-1] + 1$ $(2 \leq i \leq n)$ are non-negative. By writing $I[i]$ in the unary code $0^{I[i]}1$ to a bit vector S subsequently for each $2 \leq i \leq n$, we can compute $\mathsf{PLCP}[i] = \mathsf{select}_1(S, i) - 2i$ and $\mathsf{LCP}[i] = \mathsf{select}_1(S, \mathsf{SA}[i]) - 2\mathsf{SA}[i]$. Moreover, $\sum_{i=1}^{n} I[i] \leq n$ and therefore S is of length at most $2n$. An example is given in Fig. 3.8. The following lemma summarizes the space and time bounds:

Lemma 3.8 ([216, 237])**.** There is a data structure taking $2n + o(n)$ bits that can access LCP with SA in constant time. Having T, SA and ISA available, it can be constructed in $\mathcal{O}(n t_{\mathsf{SA}})$ time, where t_{SA} is the time to access SA.

Having a representation of SA, ISA, and LCP, the *succinct suffix tree (SST)* consists of

- ISA with $n \lg n$ bits,
- SA with $\epsilon n \lg n$ bits (using Lemma 3.7),
- LCP with $2n + o(n)$ bits (using Lemma 3.8),
- an RMQ data structure on LCP with $2n + o(n)$ bits (using Lemma 2.5 with $t_{\mathsf{SA}} = 1/\epsilon$),

- and a $4n + o(n)$-bit representation of the suffix tree topology in *depth-first unary degree sequence (DFUDS)* [29].

We construct the succinct suffix tree in the following way while making use of several algorithms from the literature:

1. SA can be constructed in $\mathcal{O}(n)$ time with $n \lg n + \mathcal{O}(\lg n)$ bits of space, including the space for SA itself ([177], or [114] in case that the alphabet is effective).

2. We invert SA to ISA with $n + \mathcal{O}(\lg n)$ bits of working space [72].

3. We construct the data structure of Lemma 3.7 to regain access to SA with $\mathcal{O}(1/\epsilon)$ access time.

4. Given SA and ISA, LCP can be computed in $\mathcal{O}(n/\epsilon)$ time with no extra space according to Lemma 3.8.

5. The RMQ data structure on LCP can be constructed in $\mathcal{O}(n/\epsilon)$ time, and answers queries in $\mathcal{O}(1/\epsilon)$ time (see Lemma 2.5).

6. Given both SA and LCP with $\mathcal{O}(1/\epsilon)$ access time, a space economical construction of the tree topology was discussed in [202, Algo. 1]. The authors showed that the DFUDS representation of the suffix tree topology can be built in $\mathcal{O}(n/\epsilon)$ time with $n + o(n)$ bits of working space.

Putting together all ingredients of the above list yields the following theorem:

Theorem 3.9. The succinct suffix tree takes $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of space. We can construct it in-place in $\mathcal{O}(n/\epsilon)$ time.

The succinct suffix tree endowed with an RMQ data structure on LCP as in Lemma 2.10 improves the result of Cor. 2.12 with respect to construction and final space.

Corollary 3.10. The succinct suffix tree from Thm. 3.9 can answer an LCE query on T in $\mathcal{O}(1/\epsilon)$ time.

The navigational methods of the suffix tree described in Sect. 3.3.3 can be answered in constant time due to the chosen suffix tree topology [136]. We can support the other methods easily with

- $\mathsf{head}(\lambda) \coloneqq T[\mathsf{sufnum}(\lambda)]$,[4]

[4] We need the text T for computing $\mathsf{head}(\lambda)$ if its alphabet Σ is not effective. If Σ is effective, we can compute $\mathsf{head}(\lambda)$ analogously to the compressed suffix tree with $\mathsf{child\_rank}(\mathsf{level\_anc}(\lambda, 1))$.

	SST	CST
Time	$\mathcal{O}(n/\epsilon)$	$\mathcal{O}(n)$
Space	$(1 + \epsilon)$ $n \lg n$ + $\mathcal{O}(n)$	$\mathcal{O}(n \lg \sigma)$

Operation	SST Time	CST Time
sufnum(λ)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(n)$
str_depth(v)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}$(str_depth(v))
edge_length(e)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}$(str_depth(v))

Fig. 3.9: *Left*: Construction time and needed space in bits for the succinct suffix tree (SST) and compressed suffix tree (CST) representations. *Right*: Time bounds for certain operations needed by our LZ factorization algorithms.

- str_depth(v) := LCP.RMQ[leaf_rank(lmost_leaf(v) + 1), leaf_rank(rmost_leaf(v))],
- smallest_leaf := leaf_select(ISA[1]), and
- next_leaf(λ) := leaf_select(sufnum(λ) + 1).

All methods using sufnum with sufnum(λ) = SA[leaf_rank(λ)] take $\mathcal{O}(1/\epsilon)$ time due to the way in which the suffix array is stored. These times differ from the query times for the compressed suffix tree – see Fig. 3.9 for a juxtaposition. As we will see in the following, our algorithms do *not* need to compute sufnum: They linearly scan over all leaves in the lexicographic order, beginning with smallest_leaf with sufnum(smallest_leaf) = 1 such that they can maintain the suffix number of currently processed leaf with a counting variable. By doing so, they can compute head(λ) and next_leaf(λ) in constant time for the currently processed leaf.

For our algorithms storing the LZ factorization within the space bounds of the succinct suffix tree, it is important that the space taken by the succinct suffix tree is *rewritable*. Let us call X the $n \lg n$ bits of space occupied by ISA, and Y the $\epsilon n \lg n$ bits space taken by SA. We chose such generic names since the contents of these arrays will change several times during the LZ77 and LZ78 computations.

Overwriting X does not only destroy ISA, but also disables the access to SA and LCP. That is because the former is represented by the data structure of Lemma 3.7 that needs access to ISA, and the latter is represented by the data structure of Lemma 3.8 that needs access to SA.

3.3.4 Framework of the LZ Algorithms

Common to our LZ77 and LZ78 factorization algorithms in Sects. 3.4 and 3.7 is the traversal of the suffix tree, during which we mark nodes. To mark nodes efficiently we uniquely identify each node of the suffix tree with its preorder number, i.e., the number induced by enumerating all nodes with a *preorder*

traversal of the suffix tree. We can address a node by its preorder number and vice versa in constant time by adding a rank- and a select-support to the bit vector representing the suffix tree topology (i.e., either in DFUDS or in the BP representation). If the context is clear, we implicitly convert a suffix tree node to its preorder number, and vice versa.

Refining this idea, we distinguish between leaves and internal nodes since we will often mark either leaves or internal nodes. Then we only have to allocate a bit vector of length n for marking either leaves or internal nodes of the suffix tree. The idea is to enumerate the leaves by their $\mathsf{leaf\_rank}$-value, and the internal nodes by their preorder numbers when omitting the leaves in the suffix tree. The latter number can be computed in constant time since we can convert the preorder number v of an internal node by $v - \mathsf{leaf\_rank}(\mathsf{lmost\_leaf}(v)) + 1$.

Next, we explain the common framework shared among our algorithms by introducing some new keywords:

Witnesses. Witnesses are *internal* nodes that act as signposts for finding (LZ77) the referred position or (LZ78) the referred index of a factor. These nodes are necessary in our approach due to the space restrictions. To compute the referred position or referred index of a factor F, we mark a certain internal node as the witness of F such that we can determine the referred position or referred index of F by examining its witness at a later stage. The *number of witnesses* z_{W} is at most the number of *referencing* factors z_{R}. We enumerate the witnesses from 1 to z_{W} using a bit vector B_{W} of length n marking internal suffix tree nodes that are witnesses. We add a rank_1-support to B_{W} such that we can map the preorder numbers of the witnesses to the B_{W}-ranks. We call $B_{\mathrm{W}}.\mathsf{rank}_1(w)$ the *witness rank* of a witness w.

Passes. We divide our algorithms into several passes. In a pass, we visit the leaves of the suffix tree in text order. This is done by taking $\mathsf{smallest\_leaf}$ and then calling $\mathsf{next\_leaf}$ successively. The passes differ in how a leaf is processed. While processing a leaf λ, we want to access $\mathsf{sufnum}(\lambda)$. We can track the suffix number of the current leaf with a counter variable, since we start at the leaf with suffix number 1.

Corresponding Leaves. We say that a leaf λ *corresponds to* a factor F if $\mathsf{sufnum}(\lambda)$ is the starting position of F. During a pass, we keep track of whether a visited leaf corresponds to a factor. This is done as follows: While processing a leaf λ corresponding to a factor F, we compute the length of F. This length tells us the number of leaves after λ (in *text order*) that do not correspond to a factor. By remembering the next corresponding leaf, we know whether the current leaf corresponds to a factor — remember that a pass selects leaves successively in *text order*, and $\mathsf{smallest\_leaf}$ always corresponds to the first factor.

Our Setting. The algorithms have to factorize (i.e., determine the factor lengths and the starting position of the factors) and compute the referred positions (LZ77) or the referred indices (LZ78). We consider two scenarios:

1. In the *first scenario*, the output has to be stored *explicitly* in RAM.

   For LZ77, we store the referred positions in an array with $z \lg n$ bits such that the x-th entry stores the referred position of the x-th factor. The factor lengths can be represented by an additional array with $z \lg n$ bits or a bit vector of length n marking the ending positions of the factors. In the latter case, we enhance the bit vector with a select-support such that we can compute the length of the x-th factor with $\mathsf{select}_1(x) - \mathsf{select}_1(x-1)$ in constant time.

   For LZ78, we store the factor indices in an array with $z \lg z$ bits. The characters at the end of the factors can be stored either

   explicitly in a $z \lg \sigma$ bits array or

   implicitly by a bit vector B_T with a select-support marking the starting positions of the factors. The bit vector B_T takes $n + o(n)$ bits and can look up the character at the end of the x-th factor ($1 \leq x \leq z$) with $T[B_T.\mathsf{select}_1(x+1) - 1]$ in constant time.[5]

   We use the *explicit* representation in conjunction with the *compressed* suffix tree, while we use the *implicit* representation in conjunction with the *succinct* suffix tree.

2. In the *second scenario (output-streaming)*, we want the output to be streamed sequentially. An output-streaming algorithm has to output a factor as a pair of values, i.e., (LZ77) its referred position and length, or (LZ78) its referred index and the character at its end; we call the set of these value-pairs the *coding* of the respective factorization (see Figs. 3.4 and 3.5). The algorithm has to output the coding sequentially *in text order*. We do not count the output in the final working space.

We can alter an output-streaming algorithm to store the factorization explicitly by adding (LZ77) $z \lg n$ bits for the referred positions and $\min(n + o(n), z \lg n)$ bits for the factor lengths, or (LZ78) $z \lg z$ bits for the referred indices and $\min(n + o(n), z \lg \sigma)$ bits for the characters at the ends of each factor.

During this chapter, we often switch between both suffix trees representations. Figure 3.10 gives a roadmap showing which suffix tree representation is used in which section.

[5] The implicit representation still requires T to be present for accessing the characters at the end of the factors. We can build the explicit representation from the implicit representation in $\mathcal{O}(z)$ time.

Section	CST	SST
LZ77 (Sects. 3.4 to 3.6)		
Sect. 3.4.1		◯
Sect. 3.4.2	◯	
Sect. 3.4.3		◯
Sect. 3.4.4	◯	◯
Sect. 3.5	◯	◯
Sect. 3.6	◯	◯
LZ78 (Sect. 3.7)		
Sect. 3.7.2	◯	
Sect. 3.7.3		◯

Fig. 3.10: Connection between the algorithms introduced in this chapter and the presented suffix tree representations. The figure shows which suffix tree representation (marked with a circle) is used by an algorithm (introduced in the respective section).

3.4 LZ77 with Space-Efficient Suffix Trees

Our LZ77 factorization algorithms factorize the text while scanning it from left to right. Suppose that we factorized the text up to the x-th factor F_x, and that the factor F_{x+1} is referencing. To determine the referred position of F_{x+1}, we need to find the longest substring $T[j\,..\,j+\ell-1]$ starting before $1+|F_1\cdots F_x|$ that is a prefix of the suffix $T[1+|F_1\cdots F_x|\,..]$. Since $T[j+\ell] \neq T[|F_1\cdots F_x|+\ell+1]$ (otherwise we could extend both substrings to the right), there is a suffix tree node w with $\mathsf{str\_depth}(w)=\ell$ having two leaves with respective suffix numbers j and $|F_1\cdots F_x|+1$. This node w is called the witness of F_{x+1}. Finding the witnesses of all referencing factors is the crucial part of our LZ77 algorithms. The witnesses can be found by the leaf-to-root traversals during a pass.

LZ77 Passes. Common to all passes is the following procedure: For each visited leaf λ, we perform a leaf-to-root traversal, i.e., we visit every node on the path from λ to the root. But we stop the leaf-to-root traversal on visiting a node visited in an earlier traversal (of the same pass). The idea is the following: When starting the leaf-to-root traversal of the leaf with suffix number j $(1 \leq j \leq n)$, the longest prefix F of $T[j\,..]$ that has an occurrence starting before j is the string label of an already visited node. That is because of the following: First, the ancestors of all leaves with suffix numbers smaller than j are already marked. Consequently, F is a prefix of the concatenation of all edge labels on the path from the root to a visited node. Second, F is actually the string label of a visited node, since otherwise we could extend F by definition of the suffix tree.

Given that the leaf with suffix number j corresponds to a factor F, the factor can be determined by accessing an already visited node in a leaf-to-root traversal from the leaf with suffix number j. We only have to access the *lowest* already visited node, since the LZ77 factorization chooses the *longest* preceding substring matching F. For this purpose we create a bit vector B_V of length n with which

we mark the visited internal nodes (we represent internal nodes by a number from 1 up to n as described in Sect. 3.3.4). This bit vector is cleared before a pass starts. Since the suffix tree of T contains at most $n-1$ internal nodes, a pass can be conducted in linear time.

LZ77 Witnesses. Determining the witnesses is done in the first pass in the following way: Let λ be a corresponding leaf. Suppose that we conduct a leaf-to-root traversal from λ. We stop the traversal at an already visited node. Reaching the root from λ corresponding to a factor (while visiting only nodes that are not marked in B_{V}) means that we found a fresh factor. Otherwise, we visit an already visited node w, where w is not the root. If λ corresponds to a factor F, w *witnesses* the referred position of F, i.e., w is the witness of λ. This means that there is a suffix starting before text position $\mathsf{sufnum}(\lambda)$ having a prefix equal to the string label of w. Moreover, w is the lowest node in the set $\{\mathsf{lca}(\lambda, \lambda') \mid \mathsf{sufnum}(\lambda') < \mathsf{sufnum}(\lambda)\}$ consisting of the LCAs of λ and all already visited leaves λ'. Consequently, the factor F corresponding to λ refers to a text position coinciding with the suffix number of a leaf belonging to w's subtree. Its length is the string length of w. Having the length of F, we can determine the starting position of the next factor, i.e., the suffix number of the next *corresponding* leaf.

We computed the witnesses for our running example in Fig. 3.11. For LZ77 (left side), the witness nodes have preorder numbers $5, 9$, and 13, and the leaves corresponding to factors have suffix numbers $1, 2, 4, 5, 8, 12$, and 15. For instance, the leaf corresponding to the fourth factor has suffix number 5. Its witness has preorder number 13. Among all descendant leaves of this witness our algorithms choose the leaf with the lowest suffix number. In this example the referred position of the fourth factor is text position 3 (that is equal to the suffix number of the chosen leaf). The length of the fourth factor is the string depth of its witness. The number of witnesses and the number of referencing factors is $z_{\mathrm{W}} = 3$ and $z_{\mathrm{R}} = 4$, respectively.

3.4.1 Alphabet-Independent Output-Streaming

We build an RMQ data structure on SA to find the leaf with the smallest suffix number in the subtree rooted at a given witness in $\mathcal{O}(1/\epsilon)$ time (according to Lemma 2.5 with $t_{\mathsf{SA}} = 1/\epsilon$). This data structure allows us to output the factorization with a single pass in linear time: On visiting an already visited node v during a leaf-to-root traversal from a corresponding leaf λ, we find the leaf λ' with the smallest suffix number having v as its ancestor in $\mathcal{O}(1/\epsilon)$ time by an RMQ on SA. The suffix number of λ' is the referred position of the factor corresponding to the leaf λ, and $\mathsf{str\_depth}(v)$ is its factor length. The access to SA is the only operation that costs us $\mathcal{O}(1/\epsilon)$ time; the other operations can be executed in constant time. In total, the algorithm runs in $\mathcal{O}(n/\epsilon)$ time. The

following theorem refines this time bound:

Theorem 3.11. Given a constant $\delta > 0$, we can compute the LZ77 factorization in $\mathcal{O}(z/(\delta\epsilon)+n/\epsilon) = \mathcal{O}(n/(\delta\epsilon))$ time using $(1+\delta)n+o(n)$ bits of working space in addition to the space needed for the succinct suffix tree when output-streaming.

Proof. We build the RMQ data structure of Lemma 2.5 on SA before inverting it to ISA during the construction of the succinct suffix tree. According to Lemma 2.5, the RMQ data structure takes δn bits and answers RMQs in $\mathcal{O}(t_{\mathsf{SA}}/\delta)$ time. It can be constructed in $\mathcal{O}(t_{\mathsf{SA}}n)$ time with $o(n)$ additional bits. The construction time is $\mathcal{O}(t_{\mathsf{SA}}n) = \mathcal{O}(n)$, since SA is stored in the array X using $n \lg n$ bits and having the access time $t_{\mathsf{SA}} = \mathcal{O}(1)$.

After inverting SA to ISA in X and storing the data structure of Lemma 3.7 representing SA in the array Y using $\epsilon n \lg n$ bits, the access time to SA and the RMQ data structure worsens to $t_{\mathsf{SA}} = \mathcal{O}(1/\epsilon)$ and $\mathcal{O}(1/(\delta\epsilon))$, respectively.

Finally, we add B_{V} using n bits to our working space. Since we get by with a single pass, we do not need to maintain B_{W}. □

Corollary 3.12. We can stream the LZ77 factorization of a text of length n in $\mathcal{O}(n/\epsilon)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of space.

Proof. We use the succinct suffix tree to compute the LZ77 factorization. Its space requirement given in Thm. 3.9 dominates the space needed for the factorization algorithm given in Thm. 3.11. □

This is already an improvement over the previously best linear-time algorithm using $2n \lg n$ bits [148] for general integer alphabets.

Unfortunately, combining this algorithm with the compressed suffix tree inherently causes the need to simulate SA. Simulating SA with the compressed suffix tree still takes $\omega(1)$ time per SA access (see Lemma 3.6 for a known trade-off with respect to space and time). Hence, we cannot hope for a linear time algorithm with an approach that uses $\mathcal{O}(n \lg \sigma)$ bits and RMQs on the suffix array (which is not stored explicitly). Luckily, with a tiny trick, we can avoid this problem by making two passes as shown in the next section.

3.4.2 Alphabet-Sensitive Algorithm

We study only the output-streaming variant, for which we claim the following result:

Theorem 3.13. Given the compressed suffix tree of T, we can compute the LZ77 factorization in $\mathcal{O}(n)$ time using $2n+z \lg n+o(n)$ additional bits of working space when output-streaming. The factorization can also be stored in RAM with additional $z \lg n$ bits.

To obtain the claim of Thm. 3.13, we perform two passes:

Algorithm 2: Pass (a) of the alphabet sensitive LZ77 algorithm of Sect. 3.4.2.

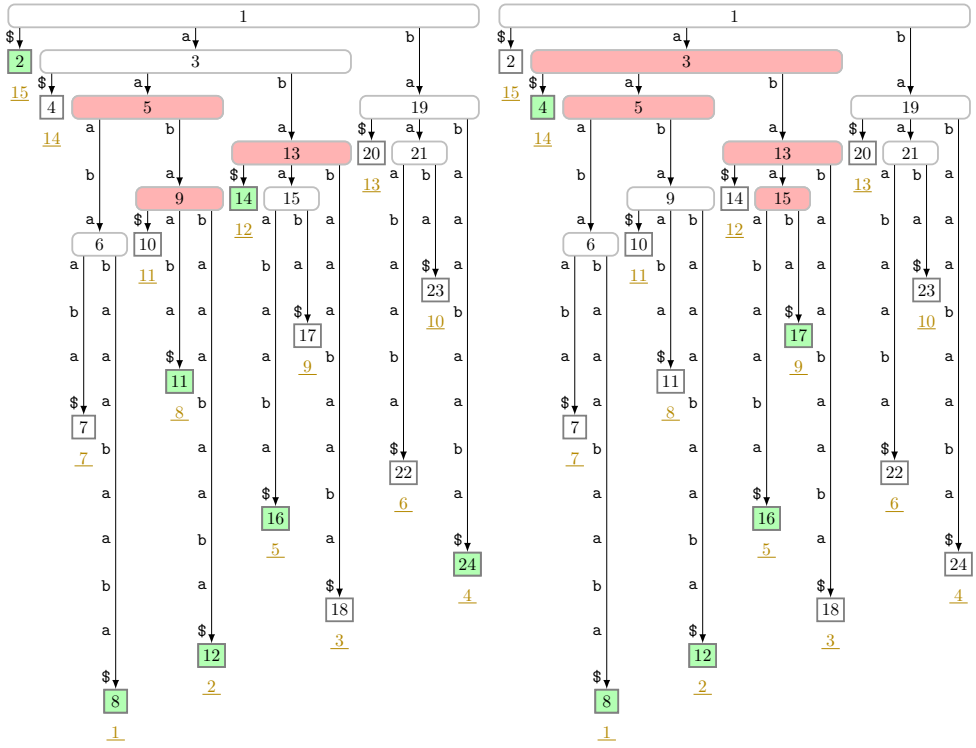
```
1  λ ← smallest_leaf
2  p ← 1                          ▷ tracks the suffix number of the next corresponding leaf
3  repeat
4      v ← parent(λ)
5      while v is not the root do
6          if B_V[v] = 1 then                                  ▷ already visited?
7              if sufnum(λ) = p then                           ▷ if λ corresponds to a factor
8                  B_W[v] ← 1                                  ▷ then v is the witness of λ
9                  p ← p + str_depth(v)                        ▷ determine the starting position of the
                                                                 next factor
10             break                                           ▷ on finding a visited node we stop
11         B_V[v] ← 1                                          ▷ mark v as visited
12         v ← parent(v)
13     if v is the root then                                   ▷ λ corresponds to a fresh factor
14         incr p                                              ▷ determine the starting position of the next factor
15     λ ← next_leaf(λ)
16 until λ = smallest_leaf
```

(a) create B_W in order to determine the witnesses (see Algo. 2), and

(b) stream the output by using an array mapping witness ranks to text positions (see Algo. 3).

Pass (a). We find the witnesses with the leaf-to-top traversals as described at the beginning of this section. On finding a witness we mark it in B_W (cf. Line 8 in Algo. 2), which helps us to determine the referred position in the next pass. After this pass, we have determined the z_W witnesses by the ones stored in B_W. We use the witnesses in the next pass to compute the referred positions (see Fig. 3.11).

Pass (b). We clear B_V, create a rank-support on B_W and allocate an array W consuming $z_W \lg n$ bits. We use W to map a witness rank to a text position (a referred position in particular, see Fig. 3.12). We set $W[w]$ to the suffix number of the leaf from which we visited the witness w for the first time (cf. Line 16 in Algo. 3). By doing so, we find the referred position of a referencing factor F in $W[w]$ on visiting w again from the leaf corresponding to F. The length of F is the string depth of w. Since fresh factors consist of single characters, we can output a fresh factor by applying head to its corresponding leaf (cf. Line 20 in Algo. 3).



Preorder	1	3	5	6	9	13	15	19	21
B_W	0	0	1	0	1	1	0	0	0

(a) LZ77 Factorization

Preorder	1	3	5	6	9	13	15	19	21
B_W	0	1	1	0	0	1	1	0	0

(b) Classic-LZ77 Factorization

Fig. 3.11: Suffix tree of Fig. 3.7 with the witness nodes and the corresponding leaves highlighted. The witness nodes and the corresponding leaves are determined during Pass (a) in Sect. 3.4.2 and Sect. 3.4.3. The witnesses are colored in red (■), the leaves corresponding to factors are colored in green (■). The number of ones in B_W is z_W.

witness rank	1	2	3
W	1	2	3

(a) LZ77

witness rank	1	2	3	4
W	1	1	3	5

(b) Classic-LZ77

Fig. 3.12: The array W storing a referred position for each witness rank in Sect. 3.4.2. The entry $W[i]$ is the suffix number of the leaf from which the i-th witness w is visited for the first time; this number is the minimum value of the suffix array in the range $[\mathsf{leaf\_rank}(\mathsf{lmost\_leaf}(w)) \,..\, \mathsf{leaf\_rank}(\mathsf{rmost\_leaf}(w))]$.

Algorithm 3: Pass (b) of the alphabet sensitive LZ77 algorithm of Sect. 3.4.2.

```
1  B_V.clear
2  B_W.add_rank_support
3  p ← 1                                  ▷ tracks the suffix number of the next corresponding leaf
4  z_W ← B_W.rank_1(|W|)
5  W ← array of size z_W lg n             ▷ maps witness ids to text positions
6  λ ← smallest_leaf
7  repeat
8      v ← parent(λ)
9      while v is not the root do
10         if B_V[v] = 1 then             ▷ invariant: B_V[v] = 1 ∧ p = sufnum(λ) ⇒ B_W(v) = 1
11             if sufnum(λ) = p then      ▷ λ corresponds to a factor
12                 output text position W[B_W.rank_1(v)]
13                 output factor length str_depth(v)
14                 p ← p + str_depth(v)   ▷ determine next starting factor
15             break
16         if B_W[v] = 1 then W[B_W.rank_1(v)] ← sufnum(λ)
17         B_V[v] ← 1
18         v ← parent(v)
19     if sufnum(λ) = p then              ▷ λ corresponds to a fresh factor
20         output character head(λ)
21         output factor length 1
22         incr p
23     λ ← next_leaf(λ)
24 until λ = smallest_leaf
```

The following corollary sums up our obtained result:

Corollary 3.14. We can compute the LZ77 factorization of a text of length n in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space.

Proof. We use the compressed suffix tree to compute the LZ77 factorization. Its space requirement given in Thm. 3.5 dominates the space needed for the factorization algorithm given in Thm. 3.13. □

Storing the Output. Instead of streaming the output we allocate an additional array with $z \lg n$ bits such that we can fill this array in Pass (b) with the referred positions. After Pass (b), we no longer need B_W and the array W. We free the space of B_W and W, but create a bit vector B_T of length n to store the factor lengths. In an additional pass, we perform the leaf-to-root traversals only

	Initial	(a)	(b)	(c)	(M)
X	ISA	ISA	ISA	D	Referred Positions
Y	SA	SA	-	-	Helper Array

Fig. 3.13: Chronological table of the contents of the arrays X and Y modified in Sect. 3.4.3. The algorithms working on the succinct suffix tree overwrite the contents of the arrays X and Y, initially storing ISA and SA. The columns represent the different phases that are chronologically sorted. Each column lists the content stored in X or Y at the respective phase.

from the corresponding leaves. Given a leaf λ corresponding to a factor F, we stop the traversal from λ when reaching the root or an already visited node w (marked in B_{V}). In the former case, F has length one. In the latter case, the already visited node w is the witness of λ. Consequently, the length of F is $\mathsf{str\_depth}(w)$, which we store in B_{T} (set $B_{\mathrm{T}}[\mathsf{sufnum}(\lambda) + \mathsf{str\_depth}(w) - 1] \leftarrow 1$).

Classic-LZ77 factorization. The LZ77 and the classic-LZ77 factorizations differ in the fact that the latter always introduces a new character at the end of each factor. We can easily adapt our algorithm to the classic-LZ77 factorization by extending each factor during the first pass, in which we determine the witnesses and factor lengths: On processing a corresponding leaf λ during this pass, we set the length of the factor F corresponding to λ to the string depth of λ's witness *incremented by one*. We can obtain the character at the end of the factor F_x when accessing the leaf whose suffix number is one text position smaller than the suffix number of the leaf corresponding to F_{x+1} (we then apply the function head on that leaf to obtain the new character). In the particular case $x = z$, the end of F_z is always \$.

We conducted the classic-LZ77 factorization on our running example in Fig. 3.11. There, the witness nodes have the preorder numbers $3, 5, 13$, and 15, and the leaves corresponding to factors have suffix numbers $1, 2, 5, 9$, and 14. We have $z_{\mathrm{W}} = z_{\mathrm{R}} = 4$.

Regarding the pseudo code of Algo. 2, we additionally need to check in Line 13 that we reach the root from a *corresponding* leaf. That is because we can reach the root from a non-corresponding leaf during a leaf-to-root traversal of the classic-LZ77 factorization. If we reach the root from a non-corresponding leaf with suffix number j, then $T[j]$ is the last character (i.e., the new character) of the last computed factor.

3.4.3 Alphabet-Independent In-Place Algorithm

Allowing only $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits for the entire working space plus output, it is no longer possible to store both SA and ISA at the same time.

Our idea is to gradually compute the necessary information about the factors (their starting positions and their lengths) such that we can overwrite X and Y when we no longer need SA and/or ISA (see Fig. 3.13 for an overview). As a final result, we will store in $X[x]$ the referred position of the x-th factor. We divide our algorithm into three passes and a final matching phase, all of which are discussed in detail in the following:

(a) Construct a bit vector $B_{\mathrm{T}}[1\,..\,n]$ marking the starting positions of all factors in T. Determine the witnesses, and mark them in B_{W}.

(b) Construct a bit vector B_{D} counting (in unary) the number of witnesses visited during each leaf-to-root traversal.

(c) Construct an array D storing the witness ranks of all witnesses visited during each leaf-to-root traversal (as counted in the second round).

(M) Convert the witness ranks in D to the referred positions.

Pass (a). This pass works exactly as Pass (a) in Sect. 3.4.2. After Pass (a), SA is no longer needed.

Before starting with the next pass, let us assume *conceptually* that each leaf λ maintains a list $\mathcal{L}_\lambda$ storing the visited witnesses during the leaf-to-root traversal from the leaf λ in chronological order. If the last visited node w during the leaf-to-root traversal from the leaf λ has been visited during a former traversal, we add w to the end of the list $\mathcal{L}_\lambda$ *only if* the leaf λ corresponds to a factor F; in this case the node w is the witness of the factor F. We call the last entry in $\mathcal{L}_\lambda$ of a corresponding leaf λ the *referred entry* of λ. The referred entry of a leaf is its witness. Given that the witness of a leaf λ is w, there is exactly one other leaf λ' maintaining a list $\mathcal{L}_{\lambda'}$ that contains w not as a referred entry, and it holds that $\mathsf{sufnum}(\lambda') < \mathsf{sufnum}(\lambda)$. The leaf λ' has also the smallest suffix number among all leaves that contain w in their lists. This means that w is first found on the leaf-to-root traversal starting from λ', and is later determined as the witness of λ. Consequently, the factor corresponding to λ refers to the text position coinciding with the suffix number of λ'. To sum up, finding the leaf whose list, excluding its referred entry, contains the referred entry of a corresponding leaf λ is the crucial step in finding the referred position of the referencing factor corresponding to λ.

To save space, we store the witnesses in the lists not by their pre-order number, but by their witness ranks. In our running example (cf. Fig. 3.11), the lists have the contents $\mathcal{L}_1 = (1)$, $\mathcal{L}_2 = (2, 1)$, $\mathcal{L}_3 = (3)$, $\mathcal{L}_4 = ()$, $\mathcal{L}_5 = (3)$, $\mathcal{L}_6 = ()$, and so on (we wrote $\mathcal{L}_{\mathsf{sufnum}(\lambda)}$ instead of $\mathcal{L}_\lambda$).

pre-order	5	9	13
witness rank	1	2	3

text position	1	2		3	5	8	12
D	1	2	1	3	3	2	3

(a) Array D

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B_{D}	1	0	1	0	0	1	0	1	1	0	1	1	1	0	1	1	1	1	0	1	1	1

(b) Bit Vector B_{D}

Fig. 3.14: The list of lists $\mathcal{L}$ represented by D (a) and B_{D} (b), computed on our running example. The number of zeros between two ones ($B_{\mathrm{D}}.\mathsf{select}_1(j)$ and $B_{\mathrm{D}}.\mathsf{select}_1(j+1)$ for a text position j with $1 \leq j \leq n$) in B_{D} equals the number of entries in D for the text position j. We shaded the referred entries of D in blue (■). Non-shaded numbers in D are unique, and each non-shaded number appears later in a referred entry at least once. The LZ77-witnesses are depicted on the left side of Fig. 3.11.

Due to Pass (a) we know that the leaf with suffix number 2 is a corresponding leaf. The list $\mathcal{L}_2$ stores at its referred entry the witness with witness rank 1. This witness is found during the leaf-to-root traversal from the leaf with suffix number 1, since this leaf is the first whose list contains this witness rank.

In the following, we aim at representing the list of lists $\mathcal{L}$, sorted by the suffix numbers of the leaves (such that $\mathcal{L}[1]$ start with the list of the leaf with suffix number 1). Despite the fact that $\mathcal{L}$ contains $z_{\mathrm{W}} + z_{\mathrm{R}}$ witness ranks in total, we want that it takes at most $n \lg n$ bits of space so that we can store it in X. This is possible due to the following lemma.

Lemma 3.15. The number of witnesses z_{W} plus the number of referencing factors z_{R} is at most n.

Proof. Let z_{R}^1 (resp. z_{R}^+) denote the number of referencing factors of length 1 (resp. longer than 1), and let z_{W}^1 (resp. z_{W}^+) denote the number of witnesses whose string depth is 1 (resp. longer than 1). Also, z_{F} denotes the number of fresh factors. Clearly, $z_{\mathrm{W}} = z_{\mathrm{W}}^1 + z_{\mathrm{W}}^+$, $z_{\mathrm{R}} = z_{\mathrm{R}}^1 + z_{\mathrm{R}}^+$, $z_{\mathrm{R}}^1 \leq z_{\mathrm{F}}$, and $z_{\mathrm{W}}^+ \leq z_{\mathrm{R}}^+$. Hence $z_{\mathrm{R}} + z_{\mathrm{W}} = z_{\mathrm{W}}^1 + z_{\mathrm{W}}^+ + z_{\mathrm{R}}^1 + z_{\mathrm{R}}^+ \leq z_{\mathrm{F}} + z_{\mathrm{R}}^1 + 2z_{\mathrm{R}}^+ \leq n$. The last inequality follows from the fact that the factors are counted disjointly by z_{F}, z_{R}^1 and z_{R}^+, and the sum over the lengths of all factors is bounded by n, and every factor counted by z_{R}^+ has length at least 2. □

The data structure storing $\mathcal{L}$ consists of (a) an integer array D storing the contents of $\mathcal{L}$, and (b) a bit vector B_{D} partitioning D into the n lists of $\mathcal{L}$. The array D can be built sequentially by appending the witness ranks whenever they are marked or referred to during the leaf-to-root traversals. The bit vector B_{D} stores a one for each text position $1 \leq j \leq n$, and intersperses these ones with zeros counting the number of witnesses written to D during the j-th traversal. In total, the bit vector B_{D} contains n ones and $z_{\mathrm{W}} + z_{\mathrm{R}}$ zeros. The ones in B_{D} implicitly divide D into n partitions (the j-th partition corresponds to the list of the leaf with suffix number j). The size of the j-th partition ($1 \leq j \leq n$) is

determined by the number of witnesses accessed during the j-th traversal. Hence the number of zeros between the $(j-1)$-th and j-th one represents the number of entries in $\mathcal{L}_\lambda$ for the leaf λ with suffix number j. Conceptually, we can access the list $\mathcal{L}_\lambda$ by $D[B_\mathrm{D}.\mathbf{rank}_0(B_\mathrm{D}.\mathbf{select}_1(j-1))+1\,..\,B_\mathrm{D}.\mathbf{rank}_0(B_\mathrm{D}.\mathbf{select}_1(j))]$, where $j = \mathbf{sufnum}(\lambda)$. Since we will perform only sequential scans over B_D, there is no need of a rank- or select-support of B_D. We depicted the bit vector B_D and the array D of our running example in Fig. 3.14. Note that D only stores witness ranks. Nevertheless, it uses $\lg n$ bits (instead of $\lg z_\mathrm{W}$ bits) per entry because we overwrite the referred entries with the respective referred positions in the end.

Finally, we show the actual computation of D and B_D. We want to store D in X to comply with the claimed space bounds. Unfortunately, up to now, we store ISA in X, which is necessary for traversing all leaves in text order, so that overwriting X naïvely with D would result in losing this functionality. This problem can be solved by performing *two* more passes, as already outlined at the beginning of Sect. 3.4.3.

Pass (b). In this pass, we compute B_D by counting the lengths of all lists in $\mathcal{L}$ with the aid of B_W. After this pass, we sparsify ISA according to B_D: We discard those ISA-values corresponding to suffixes that will not contribute to the construction of D, i.e., those values i for which there is no zero between the $(i-1)$-th and the i-th one in B_D. We align the resulting sparse ISA to the right of X.

Pass (c). We overwrite X with D from left to right using the sparse ISA. Since each suffix having an entry in the sparse ISA has at least one entry in D, overwriting the remaining ISA values before using them cannot happen.

Matching (M). Once we have D in X, we start matching referencing factors with their referred positions. Recall that each referencing factor has one referred entry, and its referred position is obtained by matching the leftmost occurrence of its witness in D.

Let us first consider the easy case with $z_\mathrm{W} \leq \lfloor n\epsilon \rfloor$ such that all referred positions fit into Y (the helper array of size $\epsilon n \lg n$ bits). In a preprocessing, we fill the entries of Y with an invalid value $\perp$. For each m with $1 \leq m \leq z_\mathrm{W}$, we use $Y[m]$ to store the suffix number of the smallest leaf among all leaves λ having the m-th witness in their list $\mathcal{L}_\lambda$.

Suppose that we have set $Y[m] = k$, i.e., the m-th witness was discovered for the first time by the traversal from the leaf with suffix number k. Whenever we read the referred entry $D[i]$ of a factor F with a starting position larger than k and $B_\mathrm{W}.\mathbf{rank}_1(D[i]) = m$, we know by $Y[m] = k$ that the referred position of F is k.

Both the filling of Y and the matching are done in one single, sequential scan over D (stored in X) from left to right: While tracking the suffix number

Algorithm 4: Matching (M) of the alphabet independent LZ77 algorithm of Sect. 3.4.3.

```
   input  : array D and bit vector B_D
   result : X[x] is referred position of the x-th factor
1  B_M ← bit vector of length |D|                    ▷ mark already processed values
2  for 0 ≤ b < ⌊z_W/nε⌋ do
3      Y[i] ← ⊥ ∀1 ≤ i ≤ nε
4      for 1 ≤ i ≤ |D| do
5          m ← B_W.rank_1(D[i])                                  ▷ 1 ≤ m ≤ z_W
6          if B_M[i] = 0 and bnε ≤ m < (b+1)nε then
7              m ← m − bnε                                       ▷ 1 ≤ m ≤ nε
8              if Y[m] = ⊥ then                                  ▷ store referred position
9                  Y[m] ← B_D.rank_1(B_D.select_0(D[i]))
10             else                                              ▷ X[i] is a referred entry
11                 X[i] ← Y[m]
12                 B_M[i] ← 1          ▷ mark that X[i] stores now a referred position
```

of the currently processed leaf with a counter k ($1 \leq k \leq n$), we look at $m \coloneqq B_{\mathrm{W}}.\mathsf{rank}_1(D[i])$ and $Y[m]$ for each array position i with $1 \leq i \leq |D|$:

- If $Y[m] = \bot$, we set $Y[m]$ to k (cf. Line 11 in Algo. 4).
- Otherwise, we have already set the value of $Y[m] \neq \bot$ to the suffix number of the first leaf λ' having the m-th witness in its list $\mathcal{L}_{\lambda'}$. Remembering that each witness can occur exactly once as a non-referred entry in a list, the value $D[i]$ is the referred entry of the factor F with starting position k (otherwise $Y[m]$ would be $\bot$). Consequently, the referred position of F is the value stored in $Y[m]$. We set $X[i] \leftarrow Y[m]$ to overwrite the referred entry of the leaf with suffix number k with the referred position of its corresponding factor F.

By doing so, we replace the witnesses stored in the referred entries of all corresponding leaves with their respective referred positions.

If $z_{\mathrm{W}} > \lfloor n\epsilon \rfloor$, we run the same scan multiple times: We partition $\{1, \ldots, z_{\mathrm{W}}\}$ into $\lceil z_{\mathrm{W}}/(n\epsilon) \rceil$ equi-distant intervals (pad the size of the last one) of size $\lfloor n\epsilon \rfloor$, and perform $\lceil z_{\mathrm{W}}/(n\epsilon) \rceil$ scans. Since each scan takes $\mathcal{O}(n)$ time, the whole computation takes $\mathcal{O}(z_{\mathrm{W}}/\epsilon) = \mathcal{O}(z/\epsilon)$ time. One problem remains: Since the domain of the witness ranks and referred positions can collide (both are integer values), we have to track which referred entries in D are already converted to referred positions. For this task, we use a bit vector B_{M} marking all processed referred entries such that we can skip the already processed referred entries. Algorithm 4 shows the pseudo code computing the matching.

Finally, we have the complete information of the factorization: The length of the factors can be obtained by a select-query on B_{T}, and X contains the referred positions of all referencing factors. By a left shift we can restructure X such that $X[x]$ gives us the referred position (if it exists) for each factor $1 \leq x \leq z$: The i-th one in B_{M} marks the position of the i-th referred positions in X. It is left to intersperse a zero for each fresh factor. This can be done by a bit vector B_{Z} of length z storing whether the x-th factor is referencing or a fresh. The bit vector can be filled within Pass (b).

The following theorem sums up the results of this section:

Theorem 3.16. Allowing the succinct suffix tree of T to be *rewritable*, we can overwrite it with the LZ77 factorization in $\mathcal{O}(n)$ time using $4n + z_{\mathrm{W}} + z_{\mathrm{R}} + z + o(n) \leq 6n + o(n)$ bits of working space on top of the space used by the suffix tree.

Proof. Besides B_{V} and B_{W} (defined in Sect. 3.4, using $2n + o(n)$ bits together) we need to allocate extra space for the bit vectors B_{D} ($n + z_{\mathrm{W}} + z_{\mathrm{R}}$ bits), B_{T} (n bits), and B_{Z} (z bits). After Pass (b), we can free the space of B_{V}, which can be used for B_{M} ($z_{\mathrm{W}} + z_{\mathrm{R}} \leq n$ bits). □

Corollary 3.17. We can compute the LZ77 factorization of a text of length n in $\mathcal{O}(n/\epsilon)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of space. The factors are stored in-place.

Proof. We create the succinct suffix tree of Thm. 3.9 to compute the LZ77 factorization. Subsequently, we overwrite the $(1+\epsilon)n \lg n$ bits of space used by the succinct suffix tree (for representing the suffix array and its inverse) in order to compute the LZ77 factorization in-place according to Thm. 3.16. □

Classic-LZ77 factorization. During the leaf-to-root traversals in Sect. 3.4.2, we enlarge each referencing factor by one (remembering that the classic-LZ77 factorization introduces an additional character at the end of each referencing factor). Since we use the bit vector B_{T} to retrieve the position and the length of a factor like before, it suffices to store the length of the factors in B_{T} according to the definition of the classic-LZ77 factorization. By doing so, the fresh character terminating a referencing factor will never be considered to be the beginning of a factor. Finally, the fresh character of each referencing factor can be looked up with B_{T} and T. Lemma 3.15 still holds for this variant of the factorization. In fact, since $z^1_{\mathrm{R}} = z^1_{\mathrm{W}} = 0$, the proof gets easier.

3.4.4 Adaptation to Computing the LPF Table

Another approach for computing the LZ77 factorization is by using the LPF table. The *longest previous factor (LPF)* table [58, 98] of T, denoted by LPF, is formally defined as $\mathsf{LPF}[j] := \max\{\ell \mid \text{there exists an } i \in [1\,..\,j-1] \text{ such that}$

$T[i\,..\,i+\ell-1] = T[j\,..\,j+\ell-1]\}$. We can construct the LZ77 factorization $T = F_1 \cdots F_z$ with it because $F_i = T[k\,..\,k+\max(1, \mathsf{LPF}[k])]$ with $k \coloneqq \sum_{j=1}^{i-1} |F_j| + 1$ for $1 \leq i \leq z$. An example is given in Fig. 3.8.

Our goal is to adapt the LZ77 algorithms of Sect. 3.4 to compute LPF in (near-linear) time with the same space bounds as for computing the LZ77 factorization. We start with the (to the best of our knowledge) state of the art algorithm with respect to time and space requirements.

Lemma 3.18 ([62, Thm. 1])**.** Given SA and LCP, we can compute LPF in $\mathcal{O}(nt_{\mathsf{SA}})$ time. Besides the output space of $n \lg n$ bits, we only need constant working space.

Apart from this algorithm, we are only aware of some practical improvements [148, 201].

With the LCP array representation of Lemma 3.8 taking $2n + o(n)$ bits, we can construct LPF with the algorithm of Crochemore et al. [62] in the following time and space bounds:

Corollary 3.19. Given SA as a plain array stored in $n \lg n$ bits (yielding $t_{\mathsf{SA}} = \mathcal{O}(1)$) and LCP stored in $2n + o(n)$ bits, we can compute LPF with $\mathcal{O}(\lg n)$ additional bits of working space (not counting the space for LPF) in $\mathcal{O}(n)$ time.

By plugging in a suffix array construction algorithm like the in-place construction algorithm by Li et al. [177], we get the bounds shown in Fig. 3.15.

Although the result of this approach seems compelling, it stores SA and LPF in plain arrays (the former for getting constant time access). In the following, we will present a more compact representation of the LPF table, for which we need the following property:

Lemma 3.20. For every integer j with $2 \leq j \leq n$, it holds that $n - j \geq \mathsf{LPF}[j] \geq \mathsf{LPF}[j-1] - 1$.

Proof. There is an i with $1 \leq i < j - 1$ such that $T[i\,..\,i + \mathsf{LPF}[j-1] - 1] = T[j-1\,..\,j-1+\mathsf{LPF}[j-1]-1]$. Hence $T[i+1\,..\,i+\mathsf{LPF}[j-1]-1] = T[j\,..\,j-1+\mathsf{LPF}[j-1]-1]$. □

We conclude that the sequence $\mathsf{LPF}[1]+1, \mathsf{LPF}[2]+2, \ldots, \mathsf{LPF}[n]+n$ is non-decreasing with $1 \leq \mathsf{LPF}[1]+1 \leq \mathsf{LPF}[n]+n \leq n$. We immediately obtain the following corollary with Lemma 3.8:

Corollary 3.21. LPF can be represented by a bit vector with a select-support such that accessing an LPF value can be performed in constant time. The data structures use $2n + o(n)$ bits.

To compute the LPF table within an improved space bound, we have to come up with a new algorithm since the algorithm of Lemma 3.18 allocates a plain array to have constant time random write access to the entries of

Algorithm	Time	Working Space	$\lvert\mathsf{LPF}\rvert$
Lemma 3.18, [62]	$\mathcal{O}(nt_{\mathsf{SA}})$	$\lvert\mathsf{SA}\rvert + \lvert\mathsf{LCP}\rvert + \mathcal{O}(\lg n)$	$n \lg n$
Cor. 3.19, [114, 177]	$\mathcal{O}(n)$	$n \lg n + 2n + o(n)$	$n \lg n$
Lemma 3.22, Thm. 3.9	$\mathcal{O}(\epsilon^{-1}n)$	$(1+\epsilon)n \lg n + \mathcal{O}(n)$	$2n + o(n)$
Lemma 3.22, Lemma 3.6	$\mathcal{O}(nt_{\mathsf{SA}})$	$\mathcal{O}(\epsilon^{-1}n \lg \sigma)$	$2n + o(n)$

Fig. 3.15: Algorithms computing LPF. The space is counted in bits. The output space $\lvert\mathsf{LPF}\rvert$ is not considered as working space. $0 < \epsilon \leq 1$ is a constant.

LPF. Luckily, we can adapt the LZ77 factorization algorithms of Sect. 3.4 to compute LPF instead of the LZ77 factorization. We aim at building the LPF table representation of Cor. 3.21 directly such that we do not need to allocate a plain array taking $n \lg n$ bits in the first place. To this end, we create a bit vector of length $2n$ and store the LPF values in it successively. We perform a single pass of one of the LZ77 factorization algorithms presented in Sect. 3.4. Similarly to the first passes of the LZ77 algorithms, we use a bit vector B_{V} to mark already visited internal nodes. On visiting a leaf we climb up the tree until reaching the root or an already marked node. In the former case (we reached the root) we output zero. In the latter case, we output the string depth of the marked node. By doing so, we have computed $\mathsf{LPF}[1\,..\,j]$ after having processed the leaf with suffix number j.

Lemma 3.22. We can compute LPF in $\mathcal{O}(n \lg_\sigma^\epsilon n)$ time with $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits of working space, or in $\mathcal{O}(n/\epsilon)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits of working space, for any constant ϵ with $0 < \epsilon \leq 1$. Both variants include the space of the output in their working spaces.

Proof. The time and space bounds are due to the construction of the suffix tree, for which we choose either the succinct suffix tree representation (Thm. 3.9) or the compressed suffix tree representation (Thm. 3.5). We compute the string depth of a node with access to an RMQ data structure of LCP, and access to SA. We can emulate both accesses with the compressed suffix tree in $\mathcal{O}(\lg_\sigma^\epsilon n)$ time according to Lemma 3.6, given that we have computed PLCP in the above representation [190, Sect. 5]. □

We present an application of Lemma 3.22 in the following section.

3.5 Application: Distinct Squares

It is well-known that a string of length n contains at most $n^2/4$ squares [93, Sect. 1]. This bound is the number of *all* squares, i.e., we count multiple occurrences of the same square, too. If we consider the number of all *distinct* squares, i.e.,

we count *exactly one* occurrence of each square, then the number becomes linear in n: The first linear upper bound was given by Fraenkel and Simpson [93] who proved that a string of length n contains at most $2n$ distinct squares. Later, Ilie [132] showed the slightly improved bound of $2n - \Theta(\lg n)$. Recently, Deza et al. [68] refined this bound to $\lfloor 11n/6 \rfloor$. In the light of these results one may wonder whether future results will "converge" to the upper bound of n: The *distinct square conjecture* [93, 138] is that a string of length n contains at most n distinct squares; this number is known to be independent of the alphabet size [184]. However, there is still a big gap between the best known bound and the conjecture. While studying a combinatorial problem like this, it is natural to think about ways to actually compute the exact number.

This section focuses on a computational problem on distinct squares, namely, we wish to compute (a compact representation of) the set of all distinct squares in a given string offline as well as online. In the offline setting, Gusfield and Stoye [121] tackled this problem with an algorithm running in $\mathcal{O}(n\sigma_T)$ time, where σ_T denotes the size of the effective alphabet of T. Although its running time is optimal $\mathcal{O}(n)$ for a constant alphabet, it becomes $\mathcal{O}(n^2)$ for a large alphabet since σ_T can be as large as n. Crochemore et al. [65] improved this result with an algorithm running in $\mathcal{O}(n)$ time for integer alphabets.

In the online setting, previous articles working with squares are due to Leung et al. [174], and Hong and Chen [125] who find the shortest prefix S of T that is a square in $\mathcal{O}(|S| \lg \sigma_T)$ time and $\mathcal{O}(|S| \lg^2 |S|)$ time, respectively, if such an S exists.[6] Unfortunately, to the best of our knowledge, there is no study on how to compute all distinct squares online.

In this section, we present an algorithm (Sect. 3.5.3) within the same time bounds of $\mathcal{O}(n)$ as [65] for integer alphabets. This algorithm is practical (Sect. 3.5.6), and can be adapted to work online (Sect. 3.5.7). Our contributions are:

- We can find all distinct squares in $\mathcal{O}(n/\epsilon)$ time with $(2+\epsilon)n \lg n + \mathcal{O}(n)$ bits of working space, or in $\mathcal{O}(nt_{\mathsf{SA}})$ time with $\mathcal{O}(n \lg \sigma)$ bits of working space (Cor. 3.27), whereas Crochemore et al. [65] achieved $\mathcal{O}(n)$ time and $\mathcal{O}(n)$ words. Next,
- we can find them *online* in $\mathcal{O}(n \lg^2 \lg n / \lg \lg \lg n)$ time using $\mathcal{O}(n)$ words of working space (Thm. 3.28),
- we can decorate the suffix tree with all distinct squares in $\mathcal{O}(n)$ time with $(\mathrm{occ} + 2n) \lg n + z \lg z + \mathcal{O}(n)$ bits of additional working space (Thm. 3.30), where occ is the number of all distinct squares,
- we can find all squares that are common to all strings of a set with a total string length of n in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ words (Thm. 3.32), and finally

[6] Otherwise, the algorithms can test whether T is square-free. For that, we substitute $|S|$ with $|T| = n$ in the running times.

i	1	2	3	4	5	6	7	8	9	10	11	12
T	a	b	a	b	a	a	a	b	a	b	a	$
SA	12	11	5	6	9	3	7	1	10	4	8	2
LCP	0	0	1	2	1	3	3	5	0	2	2	4
PLCP	5	4	3	2	1	2	3	2	1	0	0	0
LPF	0	0	3	2	1	2	5	4	3	2	1	0
LZ77	F_1	F_2	F_3			F_4		F_5				F_6

Fig. 3.16: The arrays SA, LCP, PLCP and LPF of $T =$ ababaaababa$, which is factorized in six LZ77 factors.

- we can compute the tree topology of the minimal augmented suffix tree [11] in linear time using $\mathcal{O}(n)$ words of working space (Thm. 3.33).

3.5.1 Preliminaries

Within this section, we represent the occurrences of substring S by pairs of position and length such that $S = T[i \,..\, i+\ell-1]$ for a pair (i, ℓ). A set of pairs of position and length is called *distinct*, if there are no two pairs (i, ℓ) and (i', ℓ) within this set such that $T[i \,..\, i+\ell-1] = T[i' \,..\, i'+\ell-1]$, i.e., there are no two pairs corresponding to substrings that are equal. A set of squares is a set of pairs with the restriction that each pair (i, ℓ) of this set corresponds to a square $T[i \,..\, i+\ell-1]$. A set of *all distinct squares* is a distinct set of squares that is maximal under inclusion.

Within this section, we use the text $T =$ ababaaababa$ as our running example. Figure 3.16 shows T and other support data structures needed by our algorithm computing all distinct squares.

3.5.2 Set of All Distinct Squares

Given a string T, our goal is to compute all distinct squares of T. Our algorithm represents each found square as a pair $(s, 2p)$ consisting of a starting position s and a period p such that $T[s \,..\, s+2p-1]$ is the leftmost occurrence of a square. The size of this set is linear due to the following lemma:

Lemma 3.23 (Fraenkel and Simpson [93])**.** A string of length n can contain at most $2n$ distinct squares.

We follow the approach of Gusfield and Stoye [121]. Their idea is to compute a set of squares[7] represented by pairs of starting position and length with which

[7] It differs from the set we want to compute in the fact that they allow, among others, occurrences of the same square in their set.

Fig. 3.17: Squares constructed by right rotations. This example highlights all squares of length eight with lines, where the arms of each square are separated with a small vertical bar. Squares that cannot be constructed by right-rotating are colored in dark yellow (■). The other squares are colored in blue (■). These squares can be constructed by right-rotating the square $(16, 8)$.

Fig. 3.18: Squares of our running example highlighted with lines (like in Fig. 3.17). The set of all squares is $\{(1,4), (2,4), (5,2), (6,2), (7,4), (8,4)\}$. If we take the leftmost occurrences of all squares, we get $\{(1,4), (2,4), (5,2)\}$; this set comprises all squares colored blue (■), i.e., the red lines (■) correspond to occurrences of squares that are not leftmost. In this example, the red lines form the set $\{(6,2), (7,4), (8,4)\}$, which is a set of all distinct squares. A leftmost covering set is $\{(1,4), (5,2)\}$.

they can generate all distinct squares. They call this set of squares a *leftmost covering set*. A leftmost covering set obeys the property that every square of the text can be constructed by right-rotating a square of this set. We say that a square $(k, 2p)$ is constructed by *right-rotating* a square $(i, 2p)$ with $i \leq k$ if each pair $(i+j, 2p)$ with $1 \leq j \leq k-i$ represents a square $T[i+j \,..\, i+2p+j-1] = T[i+j \,..\, i+2p-1]T[i \,..\, i+j-1]$. See Fig. 3.17 for an example.

The set of the leftmost occurrences of all squares is a set of all distinct squares. Figure 3.18 depicts an example of this set. Unfortunately, the leftmost covering set computed in [121] is not necessarily a set of all distinct squares since (a) it does not have to be distinct, and (b) a square might be missing (that can, however, be constructed by right-rotating a square of the computed leftmost covering set).

Our goal is to compute the set of all leftmost occurrences directly by modifying the algorithm of Gusfield and Stoye [121]. In what follows, we briefly review how their approach works: They compute their leftmost covering set by examining the borders between all LZ77 factors $F_1 \cdots F_z = T$. That is because of the following lemma:

Lemma 3.24 ([121, Lemma. 4])**.** The leftmost occurrence of a square cannot be contained completely in an LZ77 factor.

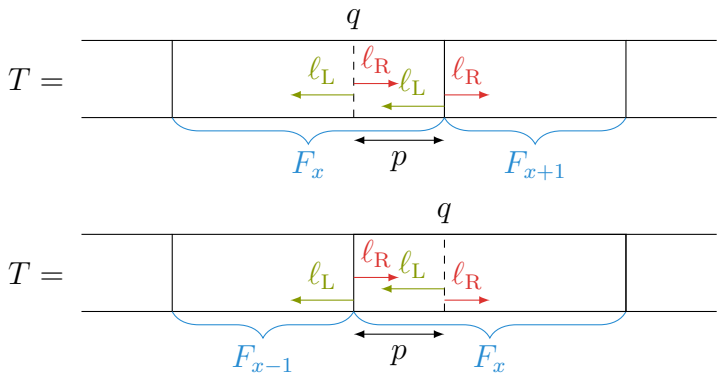


Fig. 3.19: Search for squares on the LZ77 factor borders. *Top*: Squares of Type 1 (*top*) and of Type 2 (*bottom*). Given two adjacent LZ77 factors, we determine a position q that is p positions away from the border (the direction is determined by the type of square we want to search for). By two LCE queries we can determine the lengths ℓ_L and ℓ_R that indicate the presence of a square if $\ell_L + \ell_R \geq p$.

Lemma 3.25 ([121, Thm. 5])**.** Let F_x $(1 \leq x \leq z)$ be the factor that contains the position $i + p - 1$ of the leftmost occurrence $T[i \,..\, i + 2p - 1]$ of a square. Then this occurrence belongs to one of the two types:

Type 1: its left end (position i) is inside F_x and its right end (position $i+2p-1$) is inside F_{x+1}, or

Type 2: its left end is inside an LZ77 factor preceding the x-th factor and its right end is inside F_x or F_{x+1}.

Having $\mathsf{LCE}^{\leftrightarrow}$, i.e., a data structure supporting LCE and LCS queries with $\mathcal{O}(t_{\mathrm{LCE}})$ query time, Gusfield and Stoye [121] can probe at the borders of two consecutive factors whether there is a square. Roughly speaking, they have to check at most $|F_x| + |F_{x+1}|$ many arm lengths at the borders of every two consecutive factors F_x and F_{x+1} according to Lemma 3.25 for $1 \leq x \leq z$ (set $F_{z+1} \coloneqq \Lambda$). This gives $\sum_{x=1}^{z} t_{\mathrm{LCE}}(|F_x| + |F_{x+1}|) = \mathcal{O}(nt_{\mathrm{LCE}})$ time, during which they can compute a leftmost covering set L. Figure 3.19 visualizes how the checks are done. Applying the algorithm on our running example yields the set $L = \{(1, 4), (5, 2), (7, 4)\}$. To transform this set into a set of all distinct squares, their algorithm runs the so-called Phase II that uses the suffix tree. It begins with computing the locations of the squares belonging to a subset $L' \subseteq L$ in the suffix tree in $\mathcal{O}(n)$ time. This subset L' is still guaranteed to be a leftmost covering set. Finally, the algorithm computes all distinct squares of the text by right-rotating the squares in L'. In the algorithm, the right rotations are done by *suffix link walks* over the suffix tree. The running time analysis is based on the fact that each node has at most σ_T incoming suffix links, where σ_T denotes the number of different characters occurring in the text T. Since the number of

distinct squares is linear (recall Lemma 3.23), Phase II runs in $\mathcal{O}(n\sigma_T)$ time. In total, the running time of the algorithm is dominated by Phase II. How to speed up this computation is the topic of the next section.

Algorithm 5: Helper functions used in Algo. 6.

```
   input  : starting position s and ending position e with s ≤ e
1  function recursive_rotate(s, e)
2      m ← LPF.RMQ[s .. e]                ▷ position with smallest entry in LPF[s .. e]
3      if LPF[m] > 2p then return          ▷ all squares already reported
4      report(m, 2p) and B[m] ← 1          ▷ B defined in Algo. 6
5      recursive_rotate(s, m − 1) and recursive_rotate(m + 1, e)
   input  : starting position s and arm length p of a square
6  function right_rotate
7      if B[s] = 1 then return
8      if LPF[s] < 2p then report(s, 2p) and B[s] ← 1
9      ℓ ← lcp(s, s + p)
10     recursive_rotate(s + 1, min(s + p − 1, s + ℓ − p))
```

3.5.3 Algorithmic Improvement

In the following, we present our improvement of the algorithm sketched in Sect. 3.5.2. To speed up the computation, we discard the idea of using the suffix links for right-rotating squares (i.e., we skip Phase II completely). Instead, we compute a list of all distinct squares directly. The modified algorithm outputs this list sorted first by the lengths (of the squares) and second by the starting positions.

First, we show that we can alter the original algorithm to output its leftmost covering set in the order described above. In our modification, we iterate over all possible arm lengths (Line 4 in Algo. 6), and search not yet reported squares at all LZ77 borders (Line 6), for each arm length. To achieve linear running time, we want to skip a factor F_x when the arm length becomes longer than $|F_x| + |F_{x+1}|$. We can do this with an array Z of $z \lg z$ bits that is zero initialized. When the currently tested arm length p exceeds $|F_x| + |F_{x+1}|$, we write $Z[x] \leftarrow \min\{y > x \mid |F_y| + |F_{y+1}| \geq p\}$ such that $Z[x]$ refers to the next factor whose length combined with the length of its succeeding factor is sufficiently large (cf. Line 12 in Algo. 6). By doing so, if $Z[x] \neq 0$, we can skip all factors F_y with $y \in [x \,..\, Z[x] - 1]$ in constant time. The value of $Z[x]$ is computed in amortized constant time: When scanning for the lowest $y > x$ with $|F_y| + |F_{y+1}| \geq p$, then we visit all positions y' with $x \leq y' \leq y$ never again, because we set $Z[x] = y > y'$. Maintaining the array Z allows us to run the modified algorithm still in linear time.

Algorithm 6: Modified Algo. 1 of [121].

```
input : LZ77 factorization F_1, ..., F_z
Let b(F_j) be the beginning position of the j-th factor, let F_{z+1} = Λ and b(F_{z+1}) := n.
Z ← array of size z lg z bits, zero initialized
m ← max(|F_1| + |F_2|, ..., |F_{z-1}| + |F_z|)
for p = 1, ..., m do
    B ← bit vector of length n, zero initialized
    for x = 1, ..., z do
        if |F_x| + |F_{x+1}| < p then
            y ← x
            while |F_y| + |F_{y+1}| < p do
                if Z[y] ≠ 0 then y ← Z[y]
                else incr y
            Z[x] ← y and x ← y
        if |F_x| ≥ p then                                   ▷ probe for squares of Type 1
            q ← b(F_{x+1}) − p                              ▷ b(T[i..j]) = i is the starting position
            ℓ_R ← lcp(b(F_{x+1}), q) and ℓ_L ← lcs(b(F_{x+1}) − 1, q − 1)
            if ℓ_R + ℓ_L ≥ p and ℓ_R > 0 then               ▷ found a square of length 2p with its right end in F_{x+1}
                s ← max(q − ℓ_L, q − p + 1)                 ▷ square starts at s
                right_rotate(s,p)                           ▷ see Algo. 5
        q ← b(F_x) + p                                      ▷ probe for squares of Type 2
        ℓ_R ← lcp(b(F_x), q) and ℓ_L ← lcs(b(F_x) − 1, q − 1)
        s ← max(b(F_x) − ℓ_L, b(F_x) − p + 1)               ▷ square starts in a factor preceding F_x
        if ℓ_R + ℓ_L ≥ p and ℓ_R > 0 and s + p ≤ b(F_{x+1}) and ℓ_L > 0 then
                                                            ▷ found a square of length 2p whose center is in F_x
            right_rotate(s,p)                               ▷ see Algo. 5
```

Next, we show that the modified algorithm still computes the same set. To see this, let us fix the arm length p (over which we iterate in the outer loop). According to Lemma 7 of [121], processing squares of Type 1 before processing squares of Type 2 (all squares have the same arm length p) produces the desired output for arm length p.

Finally, we show the modification that computes all distinct squares (instead of the original leftmost covering set). Roughly speaking, we use an RMQ data structure on LPF to filter already found squares. The filtered squares are used to determine the leftmost occurrences of all squares by right rotations. In more detail, we modify Algo. 1 of [121] by filtering the squares in the following way.[8] Given an arm length p, we mark in a bit vector B the beginning positions of all found squares with arm length p (cf. Lines 4 and 8 in Algo. 5). Before reporting a square, we discard the square if its starting position is already marked in B. Checking the marking in B is sufficient to prevent reporting a square more than once. The algorithm ensures that all right-rotated squares of an occurrence of a square beginning at a marked position are already reported.

Suppose that we search for the leftmost occurrences of all squares whose arm lengths are equal to p. Given the starting position s of a found square, we consider the square $(s, 2p)$ and its right rotations as candidates of our list: If $B[s] = 1$, then this square and its right rotations have already been reported. Otherwise, we report $(s, 2p)$ if $\mathsf{LPF}[s] < 2p$. To find the leftmost occurrences of all not yet reported right-rotated squares efficiently, we first compute the rightmost position e of the run with a period p containing the square $(s, 2p)$ by an LCE query (recall Cor. 2.11). Second, we check the interval $\mathcal{I} := [s+1\,..\,\min(s+p-1, e-2p+1)]$ for the starting positions of the squares whose LPF values are less than $2p$. This is sufficient, because

1. there can be at most $p-1$ different right-rotated squares of a square with arm length p, and
2. the last starting position of a square with arm length p within the same run is $e-2p+1$.

To find all entries of $\mathsf{LPF}[\mathcal{I}]$ with a value less than $2p$, we perform an RMQ query on LPF, finding the position j whose LPF value is minimal in $\mathcal{I}$. If $\mathsf{LPF}[j] \geq 2p$, then there is no leftmost occurrence of a square with arm length p in the considered range. Otherwise, we report $(j, 2p)$ and recursively search for the text position with the minimal LPF value within the intervals $[s+1\,..\,j-1]$ and $[j+1\,..\,\min(s+p-1, e-2p+1)]$, cf. Line 10 in Algo. 5. See Fig. 3.20 for an example. In total, the time of the recursion is bounded by twice the number of distinct squares starting in the interval $\mathcal{I}$, since a recursion step terminates if it could not report any square.

[8] In Line 6 of Algo. 1b of [121], the condition $start+k < h_1$ has to be changed to $start+k \leq h_1$. Otherwise, given the text $T =$ `abaabab$`, the algorithm would find only the square `aa`, but not `abaaba`.

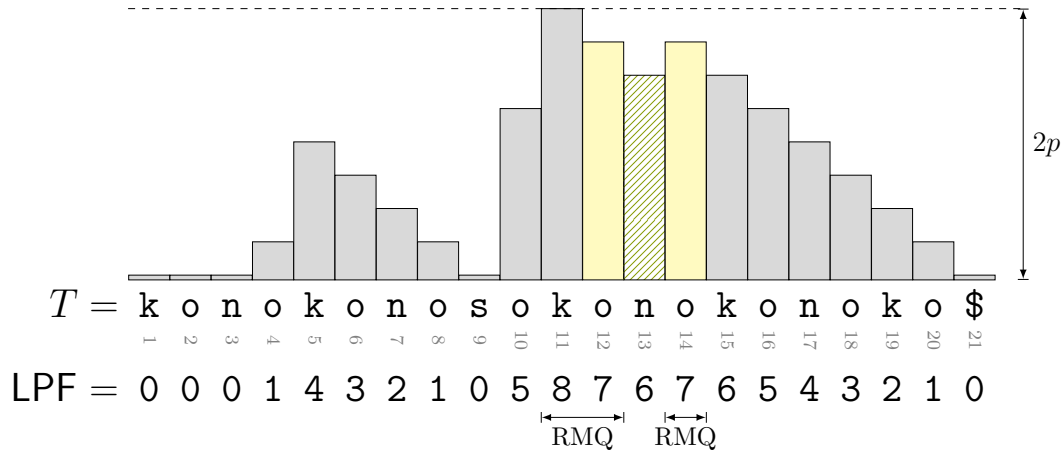


Fig. 3.20: Searching all squares with arm length 4 and a starting positions j with $\mathsf{LPF}[j] < 2p$ and $11 \leq j \leq 14$. The height of the i-th vertical bar is set in relation to the size of $\mathsf{LPF}[i]$. Suppose that we found the square `okon|okon` at position 10. To find the leftmost occurrences of those squares that are right rotations of that square, we perform recursively RMQs on LPF. The first RMQ retrieves the position whose LPF value is represented by a bar with diagonal hatching (▨). To find the starting positions (□) of the other squares, we recurse on the left and on the right side with an RMQ.

Theorem 3.26. Given $\mathsf{LCE}^{\leftrightarrow}$ with t_{LCE} access time and LPF, we can compute all distinct squares in $\mathcal{O}(nt_{\mathrm{LCE}} + \mathrm{occ}) = \mathcal{O}(nt_{\mathrm{LCE}})$ time, where occ is the number of distinct squares.

Proof. The occ term in the running time is dominated by the nt_{LCE} term due to Lemma 3.23. It is left to show that the returned list is the list of all distinct squares: No square occurs in the list twice since we only report the occurrence of a square $(i, 2p)$ if $\mathsf{LPF}[i] < 2p$. Assume that there is a square missing in the list; let $(i, 2p)$ be its leftmost occurrence. There is a square $(j, 2p)$ reported by the (original) algorithm [121] such that $i - p < j \leq i$ and right-rotating $(j, 2p)$ yields $(i, 2p)$. Since we right-rotate all found squares, we obviously have reported $(j, 2p)$. □

The next corollary, which is immediate from Thm. 3.26 and Lemma 3.22, summarizes the main result of this section.

Corollary 3.27. Given a string T of length n, we can compute all distinct squares in T in $\mathcal{O}(\epsilon^{-1}n)$ time with $(2+\epsilon)n \lg n + \mathcal{O}(n)$ bits of working space, or in $\mathcal{O}(n \lg_\sigma^\epsilon n)$ time with $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits of working space.

3.5.4 Elaborated Example

We run the algorithm devised in Sect. 3.5.3 on our running example $T =$ `ababaaababa$` step by step. The arrays SA, LCP, PLCP, LPF, and the LZ77

factorization are given in Fig. 3.16.

To ease the explanations within this example, we introduce the following definitions: We call $T[1 + |F_1 \cdots F_{i-1}|]$ (first position of the i-th factor) and $T[1 + |F_1 \cdots F_i|]$ (position after the i-th factor) the *left border* and the *right border* of F_i, respectively. The idea of the algorithm is to check the presence of a square at a factor border and at an offset value q of the border with LCE queries. The value of q is the result of either *adding* p to the *left* border, or *subtracting* p from the *right* border (see Fig. 3.19).

The algorithm finds the leftmost occurrences of all squares in the order (first) of their lengths and (second) of their starting positions. We start with $p = 1$ and try to detect squares with arm length p at each LZ77 factor border. At the beginning, we create a bit vector B marking all found squares with arm length $p = 1$. A square of this arm length is found at the right border of F_3. It is of Type 1, since its starting position is in F_3. To find it, we take the right border $b = 6$ of F_3, and the position $q \coloneqq b - p = 5$. We perform an LCE and an LCS query at b and q. Only the LCE query returns a non-zero value (here one). But this is sufficient to find the square `aa` of arm length 1. Its LPF value is smaller than $2p = 2$, so it is the leftmost occurrence. It is not yet marked in B, thus we have not yet reported it. Right rotations are not necessary for arm length 1. Having found all squares with arm length 1, we clear B.

Next, we search for squares with arm length 2. We find a square of Type 2 at the left border $b = 2$ of F_2 by performing an LCE and LCS query at b and $q \coloneqq b + p = 4$. The queries reveal that $T[1 \,..\, 5]$ is a run with a period of $p = 2$. Thus we know that $T[1 \,..\, 4]$ is a square. It is not yet marked in B, but has an LPF value smaller than $2p = 4$, i.e., it is a not yet reported leftmost occurrence. On finding a leftmost occurrence of a square, we right-rotate it, and report all right rotations whose LPF values are below $2p$. This is the case for $T[2 \,..\, 5]$, which is the leftmost occurrence of the square `baba`.

After some unsuccessful checks at the next factor borders, we come to factor F_5 and search for a square of Type 2. An LCE and LCS query at the left border $b = 8$ of F_5 and $q \coloneqq b + p = 10$ reveal that $T[7 \,..\, 11]$ is a run with a period of 2. The substring $T[7 \,..\, 10]$ is a square with $\mathsf{LPF}[7] = 5 \geq 2p$, i.e., it is a square that we have already reported. Although we have already reported it, some right rotations of it might not have been reported yet (see Sect. 3.5.5 for an example). However, this time, all right rotations (i.e., the square $T[8 \,..\, 11]$) have an LPF value of at least $2p$, i.e., there is no leftmost occurrence of a square of arm length 2 that can be found by right rotations. In total, we have found and reported the leftmost occurrences of all squares *once*.

3.5.5 Need for RMQs on the LPF Table

Remember that our algorithm computing all distinct squares performs right rotations of a square $(s, 2p)$ with recursive RMQs on the interval $\mathcal{I} \coloneqq [s + 1 \,..$

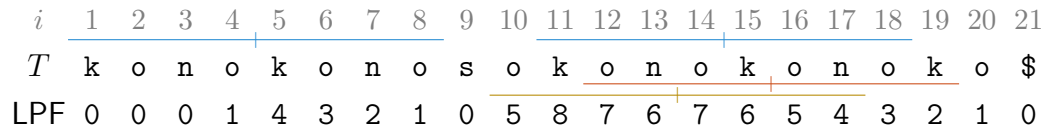


Fig. 3.21: Text of Fig. 3.20 with LPF and the squares `kono`|`kono`, `okon`|`okon`, and `onok`|`onok` highlighted with lines (the arms are separated with a vertical bar).

$\min(s+p-1, e-2p+1)]$, where e is the last position of the run with a period p that contains the square (cf. Algo. 5). Without an RMQ data structure, we could linearly scan all LPF values in $\mathcal{I}$, giving $\mathcal{O}(p) = \mathcal{O}(n)$ time. We cannot do better since the LPF values are arbitrary in general. For instance, this can be seen in the setting of Figs. 3.20 and 3.21: The square `kono`|`kono` has two occurrences starting at positions 1 and 11, whereas the square `onok`|`onok` has only one occurrence at position 12. We find this single occurrence by right-rotating the occurrence of `okon`|`okon` at position 10. We can perform the right rotation by a linear scan over LPF or an RMQ on LPF. Prepending other squares to the example can change the LPF values around that occurrence. We conclude that we cannot perform a shortcut in general (like stopping the search when the LPF value becomes at least twice as large as p). Despite the need of RMQs, the following evaluation shows that the algorithm is still useful in practice.

3.5.6 Practical Results

We have implemented the algorithm computing the leftmost occurrences of all squares[9] in C++11. The primary focus of the implementation was on the execution time, rather than on a small memory footprint: We have deliberately chosen plain 32-bit integer arrays for storing all array data structures like SA, LCP and LPF. These data structures are constructed as follows: First, we generate SA with `divsufsort`[10]. Subsequently, we generate LCP with the Φ-algorithm [146], and LPF with the simple algorithm of Crochemore et al. [62, Proposition 1]. Finally, we use the bit vector class and the RMQ data structure provided by the Succinct Data Structure Library (SDSL) [111]. In practice, it makes sense to perform an RMQ on LPF only for very large LCP values and arm lengths due to its long (yet constant) execution time. For small values, we compare characters naïvely, or scan LPF linearly.

Experimental setup. All experiments were conducted on a machine with 32 GiB of RAM, an Intel Xeon CPU `E3-1271 v3` and a Samsung SSD `850 EVO 250GB`. The operating system was a 64-bit version of Ubuntu Linux 14.04 with

[9] Available at https://github.com/koeppl/distinct_squares.
[10] Available at https://github.com/y-256/libdivsufsort.

the kernel version 3.13. All experiments ran with a single execution thread. The source code was compiled with the GNU compiler `g++ 6.2.0` using the compile flags `-O3 -march=native -DNDEBUG`.

The text collections of our experiments (here and in Sect. 3.8.2) are provided by the tudocomp framework [69, Table 1]. All texts have a length of 200 MiB. They vary in the alphabet size and in the degree of repetitiveness, e.g.,

- PC-ENGLISH is an English excerpt from the Gutenberg project[11],
- PCR-CERE is a highly-repetitive deoxyribonucleic acid (DNA) sequence with small alphabet size,
- PC-DNA is a non-highly-repetitive DNA sequence with small alphabet size, and
- HASHTAG is a tab-spaced-version data dump with integer keys and hash tags.

Figure 3.47 presents the text collections used in the evaluations.

Evaluation. Figure 3.22 shows the running times of the algorithm on the aforementioned datasets. It seems that large factors tend to slow down the computation (compare PC-ENGLISH with WIKI-ALL-VITAL), since the algorithm has to check all arm lengths up to $\max_x(|F_x| + |F_{x+1}|)$. This seems to have more impact on the running time than the number of LZ77 factors z (compare PCR-EINSTEIN.EN with PC-PROTEINS) or the number of distinct squares (compare PCR-EINSTEIN.EN with PCR-KERNEL). In most of the datasets, the running times scale worse than linear, as can be seen in Fig. 3.23.

3.5.7 Computing All Distinct Squares Online

In this section, we consider the *online* setting, where new characters are appended to the end of the text T. Given the text $T[1 \mathinner{..} i]$ up to position i with the LZ77 factorization $F_1 \cdots F_y = T[1 \mathinner{..} i]$, we consider computing the set of all distinct squares of $F_1 \cdots F_{y-2}$, i.e., up to the last two LZ77 factors. In this setting, we prove the following theorem:

Theorem 3.28. We can compute the set of all distinct squares in $\mathcal{O}(n \lg^2 \lg n / \lg \lg \lg n)$ time *online*, where n is the number of read characters. We use $\mathcal{O}(n)$ words of space during the computation.

We adapt the algorithm of Thm. 3.26 to the online setting by computing LPF online, and filling a semi-dynamic LCE data structure that answers LCE queries on the text *and* on the reversed text while supporting appending characters

[11] https://www.gutenberg.org/

Collection	$\max_x \lvert F_x F_{x+1} \rvert$	$\lvert \mathrm{occ} \rvert$	Time
HASHTAG	54 k	3160 k	196
PC-DBLP.XML	1 k	7 k	70
PC-DNA	98 k	133 k	310
PC-ENGLISH	1094 k	13 k	2639
PC-PROTEINS	68 k	3108 k	245
PC-SOURCES	308 k	340 k	792
PCR-CERE	185 k	47 k	535
PCR-EINSTEIN.EN	1634 k	18,193 k	3953
PCR-KERNEL	2756 k	9 k	6608
PCR-PARA	74 k	37 k	265
TAGME	1 k	19 k	79
WIKI-ALL-VITAL	10 k	14 k	100

Fig. 3.22: Practical evaluation of the algorithm computing all distinct squares (Sect. 3.5.6) on the datasets described in Sect. 3.5.6 and Fig. 3.47. We write 1 k for 10^3. The number of all distinct squares is $\lvert \mathrm{occ} \rvert$. Execution time is in seconds. It is the median of several conducted experiments, whose variance in time was small. The number $\max_x \lvert F_x F_{x+1} \rvert$ is the maximal length of two consecutive LZ77 factors. We needed approx. 5.73 GiB of RAM on each instance.

Collection	1 MiB	10 MiB	50 MiB	100 MiB	200 MiB
PC-DBLP.XML	0.2	3	16	33	70
PC-DNA	0.3	3	23	56	310
PC-ENGLISH	0.2	5	42	500	2639
PC-PROTEINS	0.3	4	25	74	245
PC-SOURCES	0.2	3	31	286	792
PCR-CERE	0.6	6	30	79	535
PCR-EINSTEIN.EN	0.4	12	83	1419	3953
PCR-KERNEL	0.2	8	233	1274	6608
PCR-PARA	0.4	4	26	98	265

Fig. 3.23: Computing all distinct squares on the data sets defined in Sect. 3.5.6 and Fig. 3.47. Entries are running times measured in seconds. We measured the algorithm with prefixes of 1 MiB, 10 MiB, 50 MiB, and 100 MiB of each collection.

to the text. We build this semi-dynamic LCE data structure with two online suffix tree construction algorithms. With one online suffix tree construction algorithm, we can compute LPF online and support LCE queries, while with the other we can support LCS queries.

The first is Ukkonen's algorithm that computes the suffix tree online in $\mathcal{O}(nt_{\text{nav}})$ time [236], where t_{nav} is the time for inserting a node and navigating (in particular, selecting the child on the edge starting with a specific character). We can adapt Ukkonen's algorithm to compute LPF online (similar to how Gusfield [120, Sect. 7.17.1] computes the LZ77 factorization without overlaps): Assume that we have computed the *implicit* suffix tree of $T[1\,..\,i-1]$. With *implicit* we mean that suffixes of $T[1\,..\,i-1]$ can be (not necessarily proper) prefixes of string labels of internal nodes, since the suffixes do not end with the special delimiter \$. On reading the character $T[i]$, Ukkonen's algorithm processes $T[i]$ by (1) following the suffix links of the current suffix tree, and (2) adding new leaves where a branching occurs. Suppose that it creates a new leaf with suffix number i. On adding this leaf, we additionally set $\mathsf{LPF}[i]$ to the string depth of its parent. The value of $\mathsf{LPF}[i]$ is correct, since Ukkonen's algorithm obeys two invariants: First, added leaves never change (like receiving a new suffix number). Second, on adding the leaf with suffix number i, there are already all leaves with suffix number j with $1 \leq j \leq i-1$ present in the suffix tree. Remembering the tree traversals described in Sect. 3.4, the witness of the newest added leaf i is its parent v (recall Lemma 3.22), since all other leaves in the subtree of v have a lower suffix number than i. As a consequence, the string depth of v is $\max_{1\leq j<i-1} \mathsf{lce}(i,j)$, and therefore equal to $\mathsf{LPF}[i]$. To sum up, we can update the LPF values in time linear in the update time of the suffix tree. We build the semi-dynamic RMQ data structure of Fischer [82, Sect. 3.2] (or of Ueki et al. [235, Sect. 5] if n is known beforehand) on top of LPF. This data structure takes $\mathcal{O}(n)$ words and can answer queries in constant amortized time. It can be updated also in constant amortized time.

The second suffix tree construction algorithm is a modified version [35] of Weiner's algorithm [241] that builds the suffix tree in the reversed order of Ukkonen's algorithm in $\mathcal{O}(nt_{\text{nav}})$ time. Since Weiner's algorithm incrementally constructs the suffix tree of a given text from right to left, we can adapt this algorithm to compute the suffix tree of the reversed text online in $\mathcal{O}(nt_{\text{nav}})$ time.

To get a suffix tree construction time of $\mathcal{O}(n \lg^2 \lg n / \lg \lg \lg n)$, we use the predecessor data structure of Beame and Fich [26]. We create a predecessor data structure to store the children of each suffix tree node, such that we get the navigation time $t_{\text{nav}} = \mathcal{O}(\lg^2 \lg n / \lg \lg \lg n)$ for both suffix trees. We also create a predecessor data structure to store the out-going suffix link of each node of the suffix tree constructed by Weiner's algorithm. To sum up, adding these predecessor data structures takes $\mathcal{O}(n)$ words of space in total.

Finally, our last ingredient is a dynamic LCA data structure with $\mathcal{O}(n)$ words that performs querying and modification operations in constant time [53]. The LCA of two suffix tree leaves with suffix numbers j and k is the node whose

string depth is equal to the longest common extension of $T[j\,..\,i]$ and $T[k\,..\,i]$.[12] Building this data structure on the suffix tree of the text T and on the suffix tree of the reversed text allows us to compute LCE queries in both directions in constant time.

Given the text $T[1\,..\,i] = F_1 \cdots F_y$ up to the i-th character, the entries of $\mathsf{LPF}[1\,..\,|F_1 \cdots F_{y-2}| + 1]$ are fixed (i.e., they will not change when appending new characters). To see this, we use the property that the LPF values of the leaves of the implicit suffix tree are fixed: It suffices to show that all suffixes $T[j\,..]$ are represented as leaves for each integer j with $1 \leq j \leq |F_1 \cdots F_{y-2}| + 1]$. Since $T[1\,..\,i] = F_1 \cdots F_y$ is already factorized in y LZ77 factors, the $(y-1)$-th factor cannot change due to the greedy nature of the LZ77 factorization. By definition, the substring $F_{y-1}T[1 + |F_1 \cdots F_{y-1}|]$ (which is F_{y-1} enlarged by its succeeding character) has no occurrence in T starting before F_{y-1}. Since all suffixes are prefixes of nodes in the implicit suffix tree, Ukkonen's algorithm creates a leaf for the suffix $T[1 + |F_1 \cdots F_{y-2}|\,..]$ starting with F_{y-1} on reading the character $T[1 + |F_1 \cdots F_{y-1}|]$. Remembering the aforementioned invariants of this algorithm, each leaf with suffix number j is present in the suffix tree, for $1 \leq j \leq 1 + |F_1 \cdots F_{y-2}|$.

We let the semi-dynamic RMQ data structure grow with LPF, but only up to those LPF values that are already fixed. Similarly, the text positions from 1 up to $|F_1 \cdots F_{y-2}| + 1$ are represented as leaves in both suffix trees. To sum up, our data structures support LCE queries and RMQs on LPF in the range $[1\,..\,|F_1 \cdots F_{y-2}| + 1]$ in constant time.

We adapt the algorithm of Sect. 3.5.3 by switching the order of the loops (cf. Lines 4 and 6 in Algo. 6) like in the original version [121]. The algorithm first fixes an LZ77 factor F_x and then searches for squares with an arm length between one and $|F_x| + |F_{x+1}|$. Unfortunately, we would need an extra bit vector for each arm length so that we can track all found leftmost occurrences. Instead, we use the predecessor data structure of Beame and Fich [26] storing the found occurrences of squares as pairs of starting positions and lengths. These pairs can be stored in lexicographic order (first sorted by starting position, then by length). The predecessor data structure contains at most occ elements, hence takes $\mathcal{O}(\text{occ}) = \mathcal{O}(n)$ words of space. An insertion or a search costs us $\mathcal{O}(\lg^2 \lg n / \lg \lg \lg n)$ time.

Suppose that we have computed the set for $T[1\,..\,i-1]$, and that the LZ77 factorization of $T[1\,..\,i-1]$ is $F_1 \cdots F_{y-1}$. In the case that appending a new character $T[i]$ results in a new factor F_y, we check for squares of Types 1 and 2 at the borders of F_{y-2}. Duplicates are filtered by the predecessor data structure storing all already reported leftmost occurrences. The algorithm outputs only the leftmost occurrences with the aid of LPF, whose entries are fixed up to the last two factors (this is sufficient since we search for the starting position of the

[12] Remember that we consider the text T up to the position i, hence $T[j\,..\,i]$ is (currently) the j-th suffix.

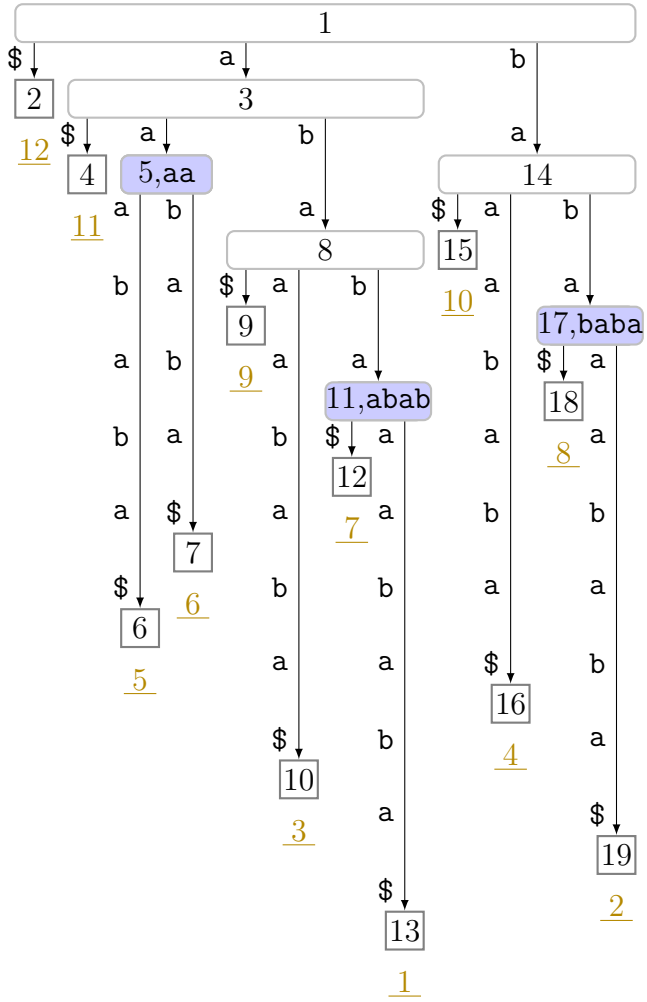


Fig. 3.24: Decorating the suffix tree of T = ababaaababa$ with all its squares. Decorated nodes are highlighted in blue (■). The label of each decorated node v is composed of v's pre-order number and the square equal to its string label.

leftmost occurrence of a square of Type 1 only in $T[1\,..\,|F_1 \cdots F_{y-2}|]$, including right rotations). In total, we need $\mathcal{O}((|F_{y-2}| + |F_{y-1}|)\lg^2 \lg n / \lg\lg\lg n)$ time.

3.5.8 Decorating the Suffix Tree with All Squares

Gusfield and Stoye described a representation of the set of all distinct squares by a decoration of the suffix tree, like the highlighted nodes (additionally annotated with its respective square) shown in the suffix tree of Fig. 3.24. This decoration has applications on its own, as we will see at the end of this section how to compute all common squares or the longest square of a set of texts. It requires a set of pairs of the form (node, length) as input such that each square $T[i\,..\,i+2p-1]$ is represented by a pair $(v, 2p)$, where v is the highest node whose string label has $T[i\,..\,i+2p-1]$ as a (not necessarily proper) prefix.

We show that we can compute this set of pairs in linear time by applying the Phase II algorithm [121] sketched in Sect. 3.5.2 to our computed set of all distinct squares. The Phase II algorithm takes a list L_i storing squares starting

at text position i, for each integer i with $1 \leq i \leq n$. Each of these lists has to be sorted in descending order with respect to the squares' lengths. It is easy to adapt our algorithm to produce these lists: On reporting a square $(i, 2p)$, we insert it at the front of L_i. By doing so, we can fill the lists *without* sorting, since we iterate over the arm length in the outer loop, while we iterate over all LZ77 factors in the inner loop.

Finally, we can conduct Phase II. In the original version, the goal of Phase II was to decorate the suffix tree with the endpoints of a subset of the original leftmost covering set. We show that exactly the same operations applied to the set of the leftmost occurrences of all squares decorate the suffix tree with all squares directly. We first augment the suffix tree leaf with suffix number i with the list L_i, for each $1 \leq i \leq n$. Subsequently, we follow Gusfield and Stoye [121] by processing every node of the suffix tree with a bottom-up traversal. During this traversal we propagate the lists of squares from the leaves up to the root: An internal node u inherits the list of the child whose subtree contains the leaf with the smallest suffix number among all leaves in the subtree rooted at u. If the edge to the parent node contains the ending position of one or more squares in the list (these candidates are stored at the front of the list), we decorate the edge with these squares, and remove them from the list. According to [121, Thm. 8], there is no square of the set L' (defined in Sect. 3.5.2) neglected during the bottom-top traversal. The same holds if we exchange L' with our computed set of all distinct squares:

Lemma 3.29. By feeding the algorithm of Phase II with the above constructed lists L_i containing the leftmost occurrences of the squares starting at the text position i, it decorates the suffix tree with all distinct squares.

Proof. We adapt the algorithm of Sect. 3.5.3 to build the lists L_i. These lists contain the leftmost occurrences of all squares. In the following we show that no square is left out during the bottom-up traversal. Let us take a suffix tree node u. Let v be the child of u whose subtree contains the leaf with the smallest suffix number among all leaves that are descendants of u. Assume, for the sake of contradiction, that there is another child w of u whose list contains the occurrence of a square $(i, 2p)$ at the time when we pass the list of v to its parent u. The length $2p$ is smaller than v's string depth, otherwise it would already have been popped off from the list. However, since v's subtree contains a leaf whose suffix number j is the smallest among the suffix numbers of all leaves contained in the subtree of w, the square occurs before at $T[j\,..\,j+2p-1] = T[i\,..\,i+2p-1]$, a contradiction to the distinctness of the set computed in Sect. 3.5.3. □

Lemma 3.29 concludes the correctness of the modified algorithm. We immediately get:

Theorem 3.30. Given LPF, $\mathsf{LCE}^{\leftrightarrow}$ with $\mathcal{O}(t_{\mathrm{LCE}})$ query time, and the suffix tree of T, we can decorate the suffix tree with all squares of the text in $\mathcal{O}(nt_{\mathrm{LCE}})$

time. Asides from these data structures, we use $(\mathrm{occ}+2n)\lg n + z\lg z + \min(n + o(n), z\lg n) + \mathcal{O}(\lg n)$ bits of additional working space.

Proof. We need $(\mathrm{occ}+2n)\lg n$ bits for storing the lists L_i ($\mathrm{occ}\lg n$ bits for storing the lengths of all squares in an integer array, and $2n\lg n$ bits for the pointers to the first element and the size of each list). The array Z uses $z\lg z$ bits. The LZ77 factors are represented in the coding described in Fig. 3.4. □

Corollary 3.31. We can compute the suffix tree and decorate it with all squares of the text in $\mathcal{O}(n/\epsilon)$ time using $(3n + \mathrm{occ} + 2n\epsilon)\lg n + z\lg z + \mathcal{O}(n)$ bits, for a constant $0 < \epsilon \leq 1$.

Proof. We use Lemma 3.22 and Thm. 3.9 to store the succinct suffix tree and LPF in $(1+\epsilon)n\lg n + \mathcal{O}(n)$ bits, supporting LCE queries with $t_{\mathrm{LCE}} = \mathcal{O}(1/\epsilon)$ according to Cor. 3.10. We additionally build the succinct suffix tree on the reversed text. Finally, we endow LPF with an RMQ data structure for the right rotations. □

All Common Squares. As an application, we consider the common squares problem: Given a set of non-empty texts with a total string length of n, we want to find all squares that occur in every string in $\mathcal{O}(n)$ time. Our main tool is the generalized suffix tree. The *generalized suffix tree* built on a set of texts is the suffix tree built on the concatenation of all texts, where we assume that each text T_j ends with a special delimiter that appears only in $T_j[|T_j|]$ and in no other text. We solve the common squares problem by first decorating the generalized suffix tree built on all texts with the distinct squares of all texts. Subsequently, we apply the $\mathcal{O}(n)$ time solution of Hui [127] that annotates each internal suffix tree node v with the number of texts that contain v's string label. This solves our problem since we can simply report all squares corresponding to nodes whose string labels are found in all texts. This also solves the problem asking for the longest common square of all texts in $\mathcal{O}(n)$ time, analogously to the longest common substring problem [120, APL4].

Theorem 3.32. Given a set of non-empty texts with a total string length of n, we can find all their common squares and the longest common square in $\mathcal{O}(n)$ time while using $\mathcal{O}(n)$ words of working space.

The following and last section is dedicated to another application of our suffix tree decoration:

3.5.9 On the Tree Topology of the MAST

A modification of the suffix tree is the *minimal augmented suffix tree (MAST)* [11]. Given a string S, the MAST of a text T can retrieve the number of all non-overlapping occurrences of S in T. The MAST differs from the suffix tree by the facts that

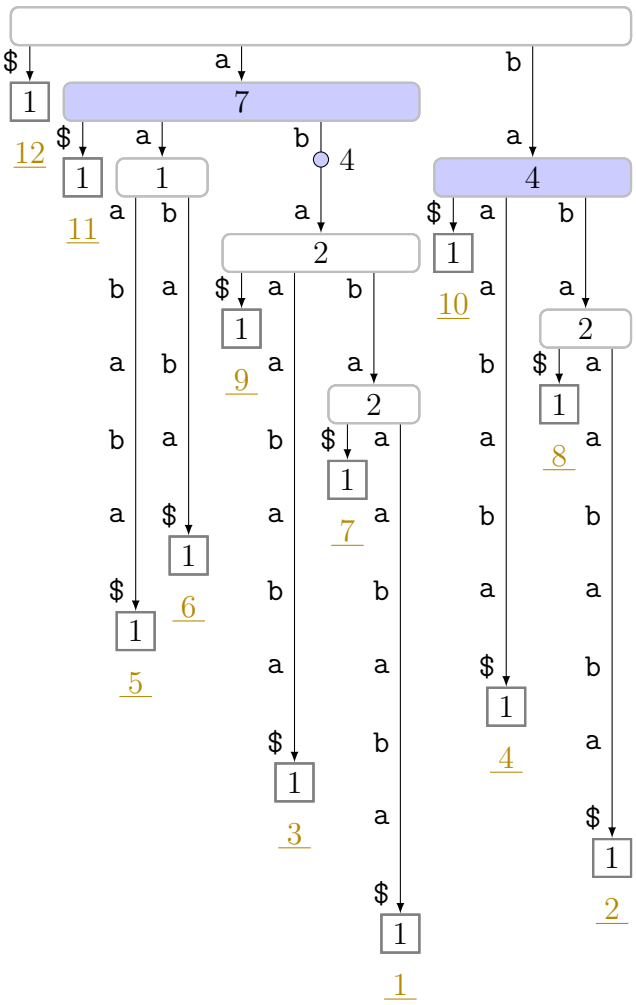


Fig. 3.25: The MAST of $T =$ `abababaababa$`. Its topology differs from the suffix tree in that it has an additional node with string label `ab`. Each node (except the root) is labeled with the number of non-overlapping occurrences of its string label in T. Nodes whose string labels are equal to the arm of a square of T are highlighted in blue (■).

- the arm of each square is the string label of a MAST node, and
- each internal node is augmented with the number of the non-overlapping occurrences of its string label.

The MAST nodes obey the property that the stored numbers of the nodes on the path from a leaf to the root are strictly increasing. Figure 3.25 shows an example.

Having the MAST, we can retrieve the number of the non-overlapping occurrences of a substring S of T in $\mathcal{O}(|S|\, t_{\text{nav}})$ time, where t_{nav} is the time to navigate from a node to one of its children. We proceed as follows: We traverse the MAST from the root downwards while reading S from the edge labels. If there is a mismatch, S cannot be found in the text. Otherwise, we end at reading the label of an edge (u, v); let u be v's parent. Then the node v is the highest node whose string label has S as a (not necessarily strict) prefix. By returning the number stored in v we are done, since this number is the number of non-overlapping occurrences of S in T.

The MAST can be built in $\mathcal{O}(n \lg n)$ time [38]. In this section, we show how to compute the tree topology of the MAST in linear time. We do this by computing a list storing the information about where to insert the missing nodes. The list stores pairs consisting of a node v and a length $2p$; we use this information later to create a new node w splitting the edge (u, v) into (u, w) and (w, v), where u is the (former) parent of v. We label (w, v) with the last $2p$ characters and (u, v) with the rest of the characters of the edge label of (u, v).

This is done as follows: We explore the suffix tree with a top-down traversal while locating the arms of the squares in the order of their lengths. To locate the arms of the squares in linear time we use two data structures. The first one is a semi-dynamic lowest marked ancestor data structure [4]. It supports marking a node and querying for the lowest marked ancestor of a node in constant amortized time. We use it to mark the area in the suffix tree that has already been processed for finding the arms of the squares.

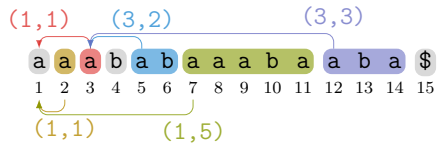
The second data structure is a list L storing pairs of the form (node, length). We compute these pairs as described in Sect. 3.5.8, where each pair $(v, 2p)$ consists of the length $2p$ of a square $T[i \mathinner{..} i + 2p - 1]$ and the highest suffix tree node v whose string label has $T[i \mathinner{..} i + 2p - 1]$ as a (not necessarily proper) prefix. We sort L with respect to the square lengths with a linear time integer sorting algorithm (e.g. Lemma 2.7).

Finally, we explain the algorithm locating the arms of all squares. We successively process all pairs of L, starting with the shortest square length. Given a pair of L containing the node v and the length $2p$, we want to split an edge on the path from the root to v and insert a new node whose string depth is p. To this end, we compute the lowest marked ancestor u of v. If u's string depth is smaller than p, we mark all descendants of u whose string depths are smaller than p, and additionally the children of those nodes (this can be done by a depth-first or a breadth-first search). If we query for the lowest marked ancestor of u again, we retrieve an ancestor w (a) whose string depth is at least p, and (b) whose parent has a string depth less than p. We report w and the subtraction of p from w's string depth (if p is equal to the string depth of w, then w's string label is the arm of the square equal to v's string label, i.e., we do not have to report it).

If the suffix tree has a pointer-based representation, it is easy to add the new nodes by splitting each edge (u, v), where v is a node contained in the output list.

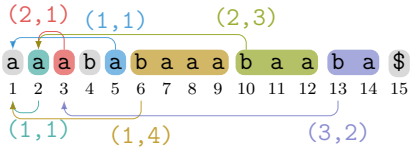
Theorem 3.33. We can compute the tree topology of the MAST in linear time using linear number of words.

Proof. By maintaining no longer needed nodes in a semi-dynamic lowest marked ancestor data structure, we visit a node as many times as we have to insert nodes on the edge to its parent, plus one. This gives $\mathcal{O}(n + 2 \operatorname{occ}) = \mathcal{O}(n)$ time, where occ is the number of all distinct squares. □



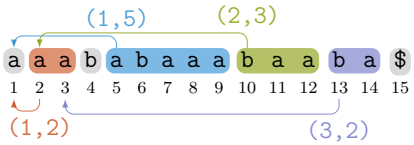
Coding: a(1,1)(1,1)b(3,2)(1,5)(3,3)$

(a) Non-Overlapping LZ77



Coding: a(1,1)(2,1)b(1,1)(1,4)(2,3)(3,2)$

(b) Non-Overlapping Reversed LZ77



(c) Overlapping Reversed LZ77

Fig. 3.26: Factorizations of Sect. 3.6 applied to our running example.

While we have seen that the LZ77 factorization is helpful for detecting squares, there are variants of the LZ77 factorization that are used for detecting other regular structures. The following section sheds a light on these factorizations:

3.6 Variants of the LZ77 Factorization

We study three variants of the LZ77 factorization, which were already considered by Crochemore et al. [64]:

- the non-overlapping LZ77 factorization (Sect. 3.6.1),
- the non-overlapping reversed LZ77 factorization (Sect. 3.6.2), and
- the overlapping reversed LZ77 factorization (Sect. 3.6.3).

Figure 3.26 illustrates each factorization applied to our running example string. Crochemore et al. [64] can compute each variant in $\mathcal{O}(n)$ time with $\mathcal{O}(n)$ words of space. For each variant, they show how to compute an LPF table adapted to the respective variant. Here, we show that we adapt our LZ77 factorization algorithms to compute each variant in at least the same time and space bounds as Crochemore et al. do.

3.6.1 Non-Overlapping LZ77

The non-overlapping LZ77 factorization is a powerful tool for finding approximate repetitions [162], periods [71], and other regular structures (see [178]). It is defined as follows (see Fig. 3.26a for an example):

Definition 3.34. A factorization $F_1 \cdots F_z = T$ is called the *non-overlapping* LZ77 of T if $F_x = \text{argmax}\{|S| \mid S \in \Sigma^* \text{ occurs in } T[1\,..\,j] \text{ or } S \in \Sigma\}$ for all $1 \leq x \leq z$ with $j = |F_1 \cdots F_{x-1}|$.

Analogously to the LPF table (see Sect. 3.4.4), there are algorithms [45, 46, 60, 64] computing the table of the longest previous non-overlapping factors, the last one [64] running in linear time. With this table, we can compute the factorization of Def. 3.34 in the same way as the LZ77 factorization with the LPF table (remember Sect. 3.4.4).

We present an adaptation of our previously introduced LZ77 factorizations, from which we borrow the notion of fresh factors, and adapt the notion of referencing factors having a referred position in the following way: The referred position of a factor $T[i\,..\,i+\ell-1]$ is the smallest text position j with $j+\ell \leq i$ and $T[j\,..\,j+\ell-1] = T[i\,..\,i+\ell-1]$. The additional restriction $j+\ell \leq i$ makes the computation of the referred positions more technical: Let j be the referred position of a factor $T[i\,..\,i+\ell-1]$, and let S be the longest substring starting before i that is a prefix of $T[i\,..]$. We associate the factor $T[i\,..\,i+\ell-1]$ with one of the following three types:

Type 1: $T[j\,..\,j+\ell-1] = S$ (this situation is the same as in the standard LZ77 variant),

Type 2: $T[j\,..\,j+\ell-1]$ is shorter than S, but $T[j+\ell] \neq T[i+\ell]$ (then there is a suffix tree node that has the string label $T[i\,..\,i+\ell-1]$), or

Type 3: $T[j+\ell] = T[i+\ell]$ (then $j+\ell = i$, otherwise the factor $T[i\,..\,i+\ell-1]$ could be extended to the right).

An example is $\mathtt{a}|\mathtt{b}|\overset{3}{\mathtt{ab}}|\overset{1}{\mathtt{a}}|\overset{2}{\mathtt{a}}|\overset{1}{\mathtt{a}}|\mathtt{\$}$, where (a) the factor borders are symbolized by the vertical bar |, and (b) the referencing factors are labeled with their types (fresh factors are not labeled). For a factor $T[i\,..]$ of Type 3, the suffixes $T[j\,..]$ and $T[i\,..]$ share more than ℓ characters such that $T[i\,..\,i+\ell-1]$ is not a string label of any suffix tree node in general, but is at least a prefix of the string label of a node. This is the case for the third factor in the aforementioned example, as can be seen in Fig. 3.27.

Despite this increased complexity, the factorization can be computed with the suffix tree in $\mathcal{O}(n \lg \sigma)$ time using $\mathcal{O}(n \lg n)$ bits of space [120, APL16]. We adapt the algorithms of Sect. 3.4 computing the overlapping LZ77 factorization to compute the non-overlapping factorization by following the approach of Gusfield [120].

Witnesses. Like in Sect. 3.4, we use witnesses to create a connection between corresponding leaves and their referred positions. The witness of a leaf λ corresponding to a factor F is the lowest node whose subtree contains λ and a leaf with suffix number j such that $T[j\,..\,j+|F|-1]$ is the largest substring

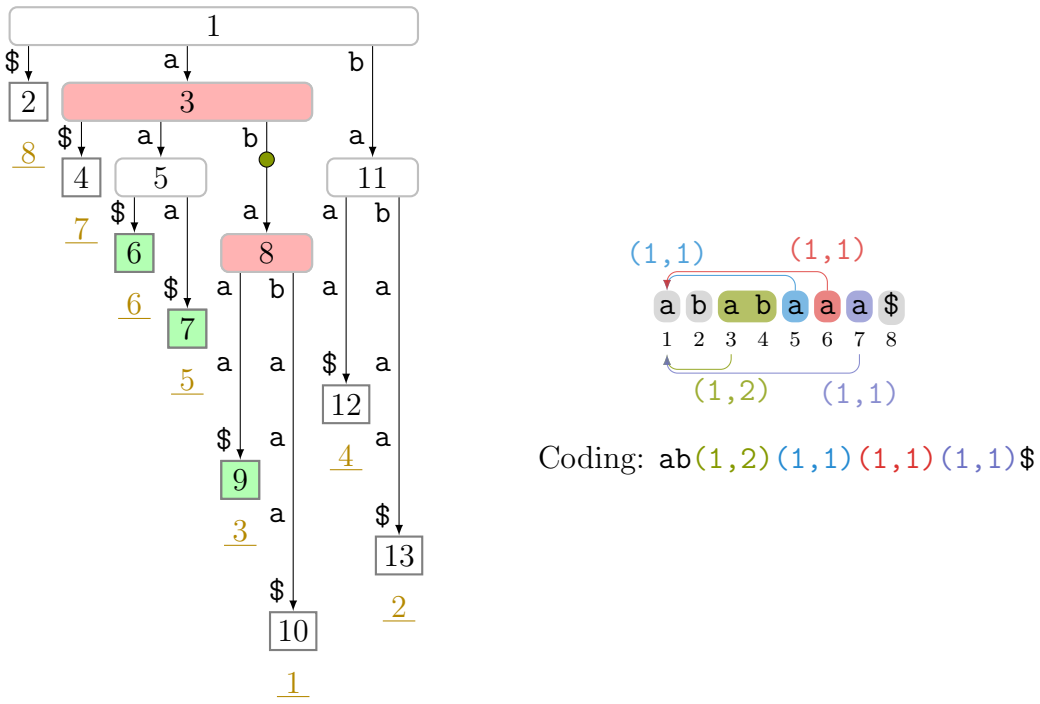


Fig. 3.27: *Left:* Suffix tree of the text $T = \texttt{ababaaa\$}$ with the witness nodes and the corresponding leaves of the non-overlapping LZ77 factorization (Def. 3.34) highlighted in red (■) and in green (■), respectively. We additionally marked the string ab with an implicit node (■) whose string label is equal to the factor with Type 3. *Right:* Non-overlapping LZ77 factorization of T.

of $T[1\,..\,\mathsf{sufnum}(\lambda) - 1]$ that is a prefix of $T[\mathsf{sufnum}(\lambda)\,..]$ (see also Figs. 3.27 and 3.28). Then j is the referred position of F (we always select the smallest such j).

In what follows, we present our modification, which solely consists of a modification of Pass (a). Pass (b) of Sect. 3.4.2, or Pass (b) and Pass (c) of Sect. 3.4.3 are identical to the standard variants (e.g., marking visited nodes in B_V). In Pass (a) of all LZ77 algorithms, we determine the witnesses. The goal of Pass (a) is either to output the coding directly (in the streaming-output variant), or to mark and store the witnesses in a bit vector B_W. When working in multiple passes with the succinct suffix tree, we additionally have to store information about the witnesses to find them later. That is because we determine them in our modification with the suffix array, which we delete in the following passes of Sect. 3.4.3 to free up space.

Pass (a). Instead of performing leaf-to-root traversals, we traverse from the root to a specific leaf. We perform such a traversal by level ancestor queries such that visiting a node takes constant time. We perform these traversals only for all *corresponding* leaves since the other leaves are not useful for determining

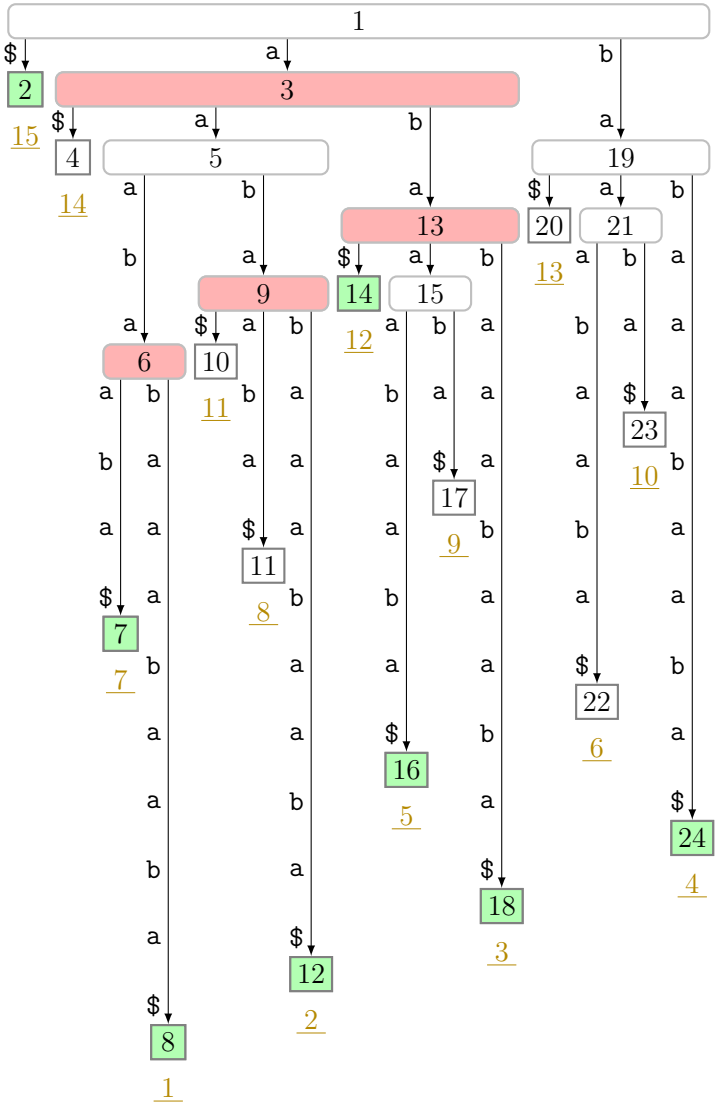


Fig. 3.28: Suffix tree of $T =$ `aaababaaabaaba$` with the witness nodes and the corresponding leaves of the non-overlapping LZ77 factorization (Sect. 3.6.1) highlighted in red (■) and green (■), respectively.

a factor. Given a node v on the path from the root to a leaf λ corresponding to a factor F, let j_v be the smallest suffix number among all leaves belonging to v's subtree. Further let $\mathcal{I}_v := [j_v \,..\, j_v + \textsf{str\_depth}(v) - 1]$ and $\mathcal{I}_{\lambda,v} := [\textsf{sufnum}(\lambda) \,..\, \textsf{sufnum}(\lambda) + \textsf{str\_depth}(v) - 1]$ be two intervals. These two intervals have the property that $T[\mathcal{I}_v] = T[\mathcal{I}_{\lambda,v}]$. We compute the values of j_v, $\mathcal{I}_v$ and $\mathcal{I}_{\lambda,v}$ for every node v on the path from the root to λ until reaching a node v such that the intervals $\mathcal{I}_v$ and $\mathcal{I}_{\lambda,v}$ overlap (cf. Line 9 in Algo. 7). Let u be the parent of v. Then the edge (u, v) determines the factor F: We consider the following two cases that determine whether F is a fresh or referencing factor, and whether the witness and the referred position of F are u and j_u, or v and j_v, respectively, in case F is referencing:

- If $j_v = \textsf{sufnum}(\lambda)$, there is no leaf in v's subtree with a suffix number smaller than $\textsf{sufnum}(\lambda)$.

  - If u is the root, then there is no candidate for a referring position available, i.e., F is a fresh factor (cf. Line 13 in Algo. 7).
  - Otherwise, $\mathsf{str\_depth}(u) > 0$ and $\mathcal{I}_u \cap \mathcal{I}_{\lambda,u} = \emptyset$ (since we reached the edge (u, v) instead of the previous edge $(\mathsf{parent}(u), u)$ on the path from the root to λ). Hence, the longest substring occurring before $\mathsf{sufnum}(\lambda)$ that is a prefix of $T[\mathsf{sufnum}(\lambda)\,..]$ has an occurrence in $T[1\,..\,\mathsf{sufnum}(\lambda) - 1]$ (Type 1). One of those occurrences starts at position j_u. This means that the referred position is j_u, and the witness of λ is u; the length of F is $\mathsf{str\_depth}(u)$ (cf. Line 17 in Algo. 7).

- If $j_v < \mathsf{sufnum}(\lambda)$, the length of F is in the interval $[\mathsf{str\_depth}(u)\,..\,\mathsf{str\_depth}(v) - 1]$. If the factor F refers to the position j_v, then its length is the minimum of $\mathsf{sufnum}(\lambda) - j_v$ and the length of the LCP of the suffixes starting at j_v and $\mathsf{sufnum}(\lambda)$. Let us denote the value of this minimum by ℓ, which determines whether F refers to j_v or j_u:
  - If $\ell = \mathsf{str\_depth}(u)$, then the referred position of F is actually the suffix number of a leaf contained in u's subtree (Type 2). In this case, the length of F is $|\mathcal{I}_u| = \mathsf{str\_depth}(u)$ because $\mathcal{I}_u \cap \mathcal{I}_{\lambda,u} = \emptyset$. The witness of λ is u, and j_u is the referred position (cf. Line 21 in Algo. 7).
  - Otherwise, $\mathsf{str\_depth}(u) < |F| < \mathsf{str\_depth}(v)$, hence F is not the string label of any suffix tree node (Type 3). The node v is the highest node whose string label has F as a prefix. We conclude that the witness, the referred position and the length of F are v, j_v, and ℓ, respectively (cf. Line 23 in Algo. 7).

To determine the value of j_v, we require an RMQ data structure on SA (similar to Thm. 3.11). According to Lemma 2.5, such a data structure can be constructed in $\mathcal{O}(t_{\mathsf{SA}} n)$ time. A query needs access to SA and costs t_{SA} time. We can access SA in $\mathcal{O}(1/\epsilon)$ time and in $\mathcal{O}(\lg^{\epsilon}_{\sigma} n)$ time with the succinct and the compressed suffix tree (with Lemma 3.6), respectively. Since the number of visited nodes is at most the factor length of a corresponding leaf λ during a root-to-leaf traversal to λ, and $\sum_{x=1}^{z} |F_x| = n$, we conclude that the RMQ queries take $\mathcal{O}(n t_{\mathsf{SA}})$ time in total, where t_{SA} is the time to access SA.

For each root-to-leaf traversal to a leaf corresponding to a factor F, we stop at an edge (u, v), and compute the length of the LCP of $T[j_v\,..]$ and $T[\mathsf{sufnum}(\lambda)\,..]$ by naïvely comparing $\mathcal{O}(|F|)$ characters. The number of compared characters is $\mathcal{O}(\sum_{x=1}^{z} |F_x|) = \mathcal{O}(n)$. Altogether, a pass takes $\mathcal{O}(n t_{\mathsf{SA}})$ time, since all applied tree navigational operations take constant time.

In the output-streaming scenario, we can directly output the referred position and the length of the factor corresponding to λ. In the other scenario, we need to store additional information for retrieving the witness w in a later pass, since

Algorithm 7: Pass (a) of the non-overlapping LZ77 algorithm of Sect. 3.6.1. The function report(w, j, ℓ) either outputs the referred position j and the length ℓ of the respective referencing factor, or marks the witness w and the starting position of the next factor (determined by ℓ) in B_{W} and in B_{T}, respectively, and appends the unary value of depth(w) to B_{L}.

```
1  λ ← smallest_leaf                                   ▷ invariant: λ is always corresponding leaf
2  repeat
3      d ← 1                                           ▷ depth counter for level_anc(λ, d)
4      ℓ ← 0                                           ▷ length of the factor corresponding to λ
5      while d ≠ depth(λ) do
6          v ← level_anc(λ, d)
7          j_v ← SA[SA.RMQ[leaf_rank(lmost_leaf(v)),
                          leaf_rank(rmost_leaf(v))]]
                                  ▷ j_v is the smallest suffix number of all leaves of v's subtree
8          ℓ ← str_depth(v) − 1
9          if [j_v .. j_v + ℓ] ∩ [sufnum(λ) .. sufnum(λ) + ℓ] = ∅ then
10             incr d
11             continue
12         u ← parent(v)
13         if j_v = sufnum(λ) then          ▷ λ has smallest suffix number in v's subtree
14             if u is the root then                    ▷ λ corresponds to a fresh factor
15                 ℓ ← 1 and output fresh factor
16                 break
17             ℓ ← str_depth(u)                                              ▷ Type 1
18             report(u, j_u, ℓ)          ▷ j_u was already computed in previous iteration
19             break
20         ℓ ← min(lce(j_v, sufnum(λ)), sufnum(λ) − j_v)
21         if ℓ ≤ str_depth(u) then                                          ▷ Type 2
22             ℓ ← str_depth(u) and report(u, j_u, ℓ)
23         else report(v, j_v, ℓ)                                            ▷ Type 3
24         break
25     λ ← next_leaf^(ℓ)(λ)                  ▷ move to the next corresponding leaf
26 until λ = smallest_leaf
```

SA (and hence the RMQ data structure built on SA) is only available in Pass (a) when working with the succinct suffix tree. Although we mark each witness in the bit vector B_W, there can be multiple nodes marked in B_W on the path from the root to a corresponding leaf. In the standard LZ77 factorization we take the invariant for granted that the witness of a leaf λ is the lowest ancestor of λ that is marked in B_V, given that B_V marks all ancestors of the leaves with a suffix number smaller than $\mathsf{sufnum}(\lambda)$. Due to the existence of factors of Types 2 and 3, this invariant does not hold for the non-overlapping factorization.

For the later passes, we want a data structure that finds the witness w of a leaf λ based on $\mathsf{sufnum}(\lambda)$ in constant time. Fortunately, w is determined by λ and its depth due to $w = \mathsf{level\_anc}(\lambda, \mathsf{depth}(w))$. To remember the depth of each witness, we maintain a bit vector B_L that stores the depth of each witness in unary coding sorted by the suffix number of the respective corresponding leaf. Given that we find the witness w of a leaf λ in Pass (a) during the traversal from the root to λ, we store the unary code $\mathtt{0}^d\mathtt{1}$ in B_L, where $d = \mathsf{depth}(w)$. For a leaf corresponding to a fresh factor, we store the unary code $\mathtt{1}$ in B_L. Like B_D in Pass (b) of Sect. 3.4.3, we do not need to add a select-support to B_L, since we process the corresponding leaves always sequentially in text order. Given a corresponding leaf λ, we can jump to its witness (or to the root if it corresponds to a fresh factor) with a level ancestor query from λ with the depth $B_L.\mathsf{select}_1(\mathsf{sufnum}(\lambda) + 1) - B_L.\mathsf{select}_1(\mathsf{sufnum}(\lambda)) - 1$.

With Lemma 3.6 we finally obtain:

Theorem 3.35. We can compute the non-overlapping LZ77 factorization

- in $\mathcal{O}(\epsilon^{-1}n)$ time using $(1+\epsilon)n \lg n + \mathcal{O}(n)$ bits, or
- in $\mathcal{O}(n \lg^{\epsilon}_{\sigma} n)$ time using $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits.

3.6.2 Non-Overlapping Reversed LZ77

The non-overlapping reversed LZ77 factorization was introduced by Kolpakov and Kucherov [163]. The authors introduced this factorization as a helpful tool for detecting gapped palindromes. It is defined as follows:

Definition 3.36. A factorization $F_1 \cdots F_z = T$ is called the *non-overlapping reversed LZ77 factorization* of T if

$$F_x = \operatorname{argmax}\{|S| \mid S \in \Sigma^* \text{ occurs in } T[1\,..\,j]^{\mathsf{T}} \text{ or } S \in \Sigma\}$$

for all $1 \leq x \leq z$ with $j = |F_1 \cdots F_{x-1}|$.

The reversed LZ77 factorization can be computed in $\mathcal{O}(n \lg \sigma)$ time using $\mathcal{O}(n \lg n)$ bits of space [163]. There is an online algorithm running in $\mathcal{O}(n \lg^2 \sigma)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space [225]. The factorization can be computed with a table storing the longest previous non-overlapping reverse factor for each text

position, like the LPF table (see Sect. 3.4.4). There are algorithms [18, 43, 44] computing this table in linear time for strings whose characters are drawn from alphabets with constant sizes, which got finally generalized by Crochemore et al. [64] and Dumitran et al. [70, Thm. 1] to integer alphabets. We present an alternative approach based on our algorithms of Sect. 3.4. Our approach runs with the same space and time bounds as [64]. Since it needs $\mathcal{O}(n)$ words of space, we can use either the succinct suffix tree or the pointer-based representation.

Like for the standard LZ77 factorization, the factors of the reversed LZ77 factorization are coded either as fresh factors or as referencing factors. Given that the x-th factor is referencing, its referred position j must end before the text position $1+|F_1 \cdots F_x|$ such that $T[j\,..\,j+|F_x|-1]^\intercal = T[|F_1 \cdots F_{x-1}|+1\,..\,|F_1 \cdots F_x|]$. To find this position, we propose an approach based on the suffix tree of the concatenation $T[1] \cdots T[n]T[n-1] \cdots T[1]\# = T(T[1\,..\,n-1])^\intercal\#$, where $\# \notin \Sigma \cup \{\$\}$ is a new character that is smaller than every character in $\Sigma \cup \{\$\}$. With this suffix tree we conduct the passes like for the standard LZ77 factorization. We use again the concept of witnesses, adapted to the reversed LZ77 factorization:

Witnesses. A witness w of a leaf λ corresponding to a *referencing* factor is the LCA of λ and a leaf with suffix number $2n-j$ $(1 \le j \le n-1)$ such that $T[j\,..\,j+\mathsf{str\_depth}(w)-1]$ is the longest substring in $T[1\,..\,\mathsf{sufnum}(\lambda)-1]$ that is a suffix of $T[\mathsf{sufnum}(\lambda)\,..]^\intercal$. The smallest such j is the referred position of λ that we want to compute.

Passes. Similarly to Sect. 3.4, we greedily parse the factors from left to right in T. The difference is that we maintain all possible referred positions by scanning the reversed text from right to left. We scan the reversed text sequentially by processing the leaves with suffix numbers $n+1, \ldots, 2n-1$ of the suffix tree in *reversed* text order while marking visited nodes in a bit vector B_V. We can conduct a pass in reverse text order by exchanging $\mathsf{next\_leaf}$ with $\mathsf{prev\_leaf}$, where $\mathsf{prev\_leaf}(\lambda)$ returns the leaf with suffix number $\mathsf{sufnum}(\ell)-1$ (or the leaf with the largest suffix number in case that $\lambda = \mathsf{smallest\_leaf}$). The idea is to actually perform two passes simultaneously: We mingle the processing of the leaves with suffix number $1, \ldots, n-1$ in text order with the processing of the leaves with suffix number $n+1, \ldots, 2n-1$ in reverse text order. After processing a leaf with suffix number i $(1 \le i \le n-1)$, we subsequently process the leaf with suffix number $2n-i$. The leaves with a suffix number *smaller* than n can be corresponding leaves, whose suffix numbers are the starting positions of the factors.

Pass (a). In this pass, we determine the witnesses with leaf-to-root traversals (see Fig. 3.29 for an example and Algo. 8 for the pseudo code). We create the bit vector B_V marking the nodes that are visited during the leaf-to-root

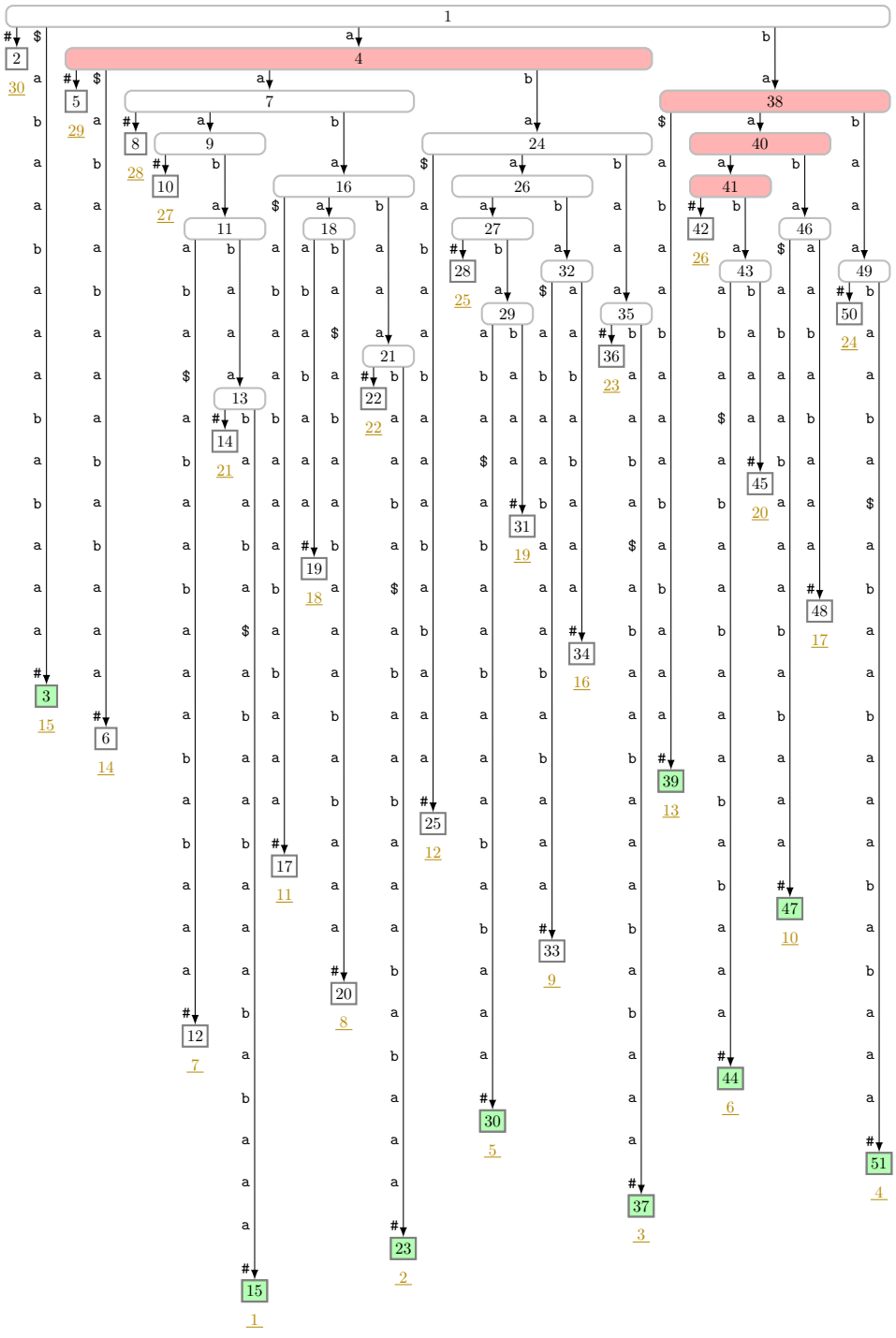


Fig. 3.29: Suffix tree of $T(T[1\,..\,n-1])^{\intercal}\#$ used in Sect. 3.6.2, where $T =$ `aaababaaabaaba$`. The witnesses are colored in red (■), the leaves corresponding to factors are colored in green (■).

traversal from each leaf with suffix number $n+1, \ldots, 2n-1$. Initially, we mark the root in B_{V}. We use the variables λ and $\lambda^{\intercal}$ pointing to the leaves that we currently process in text order and in reverse order, respectively. Given that we currently process text position i for $1 \leq i \leq n-1$, the leaves λ and $\lambda^{\intercal}$ have suffix numbers i and $2n-i$, respectively. At this point, the ancestors of the leaves with suffix numbers $2n-i+1, \ldots, 2n-1$ are already marked in B_{V}. We start with processing λ, and then subsequently process $\lambda^{\intercal}$. If λ is not a corresponding leaf, we skip it. Otherwise, given that λ is corresponding to a factor F, we perform a leaf-to-root traversal from λ (without marking visited nodes). We stop the traversal on visiting a node w that is marked in B_{V}. If w is the root, then F is a fresh factor. Otherwise, w has already been visited during a leaf-to-root traversal from a leaf with suffix number $2n-j$ $(1 \leq j \leq i-1)$. The smallest possible value for j is the referred index of λ. Consequently, the witness and the length of F are w and $\mathsf{str\_depth}(w)$, respectively. Whether or not λ is a corresponding leaf, we subsequently perform a leaf-to-root traversal from $\lambda^{\intercal}$, and mark all visited nodes during this traversal in B_{V}. By doing so, we make the text position i a candidate for a referred position of the succeeding factors. Finally, we select the leaf succeeding λ and the leaf preceding $\lambda^{\intercal}$ with respect to text order, and continue iterating until $\lambda = \lambda^{\intercal}$ (that is the case when $\mathsf{sufnum}(\lambda) = n$). In this case we output the fresh factor $F_z = \$$, and are done.

Having a range *maximum* query data structure on SA available, we can directly output the factorization in one pass as in Sect. 3.4.1 (cf. Line 10 in Algo. 8). Otherwise, we only mark the witnesses and the factor lengths in bit vectors B_{W} and B_{T}, respectively, like in Pass (a) of the multi-pass algorithms. For the other passes, we follow the same approach as in the multi-pass algorithms to determine the referred positions.

Unfortunately, a node can be visited $\Omega(n^2)$ times since we do not mark nodes in B_{V} during the leaf-to-root traversals from the leaves with suffix numbers $1, \ldots, n-1$ (an example is $T = \mathtt{a}^n$). To obtain linear time, we use a marked ancestor data structure:

Theorem 3.37. We can compute the reversed LZ77 factorization in linear time with $\mathcal{O}(n)$ words.

Proof. We augment B_{V} with a marked ancestor data structure (see [4, Table 1] or [7, Sect. 3.1]) such that a marking of a node in B_{V} marks the respective node in this data structure. The data structure supports marking and jumping to the lowest marked node from a leaf in $\mathcal{O}(1)$ amortized time. We use this data structure to directly jump to a marked node in each leaf-to-root traversal from the corresponding leaves that are processed in text order. The leaf-to-root traversals in reverse text order take linear time overall because these leaf-to-root traversals stop on visiting an already visited node. □

3.6.3 Overlapping Reversed LZ77

Last but not least, Crochemore et al. [64] and Sugimoto et al. [225] also considered the overlapping reversed LZ77 factorization:

Definition 3.38. A factorization $F_1 \cdots F_z = T$ is called the *overlapping reversed LZ77 factorization* of T if $F_x = \text{argmax}\{|S| \mid S \in \Sigma^* \text{ is a suffix of } T[i\,..]^\intercal \text{ for an } i \text{ with } 1 \leq i \leq |F_1 \cdots F_{x-1}|, \text{ or } S \in \Sigma\}$ for all $1 \leq x \leq z$.

Unfortunately, this factorization cannot be expressed in a compact LZ77 coding that stores enough information to restore the original text. To see this, take a palindrome P, and compute the overlapping reversed LZ77 factorization of $\mathtt{a}P\mathtt{a}$. The factorization creates the two factors $\mathtt{a}$ and $P\mathtt{a}$. The second factor refers to the first text position. Its reverse is $\mathtt{a}P = (P\mathtt{a})^\intercal$. However, a coding of the second factor needs to store additional information about P to support restoring the characters of this factor.

In the following, we show that we can compute this factorization in linear time (but without a coding). The idea is to adapt the algorithm in Sect. 3.6.2 computing the non-overlapping reversed LZ77 factorization for our problem. Each time it determines a factor, we try to extend the factor if it starts within a maximal palindrome. To know the locations of the maximal palindromes within T, we compute them in a precomputation step with Lemma 2.4. We store the starting position $\mathsf{b}(P)$ and the length $|P|$ of each maximal palindrome P as a pair in a list L, and sort L with respect to the starting positions with a linear-time integer sorting algorithm (e.g., with Lemma 2.7).

The actual computation performs exactly one pass, as described for the non-overlapping variant. Additionally to the leaf λ, we maintain the palindrome P of L whose intersection with the segment $T[\mathsf{sufnum}(\lambda)\,..]$ is the largest among all other palindromes (set P and $\mathsf{b}(P)$ to Λ and 0, respectively, if no palindrome intersects with $T[\mathsf{sufnum}(\lambda)\,..]$). Given that the non-overlapping reversed LZ77 factorization creates a factor shorter than $\mathsf{b}(P) + |P| - \mathsf{sufnum}(\lambda)$, we extend the factor to $T[\mathsf{sufnum}(\lambda)\,..\,\mathsf{b}(P) + |P| - 1]$. We can do that because $T[\mathsf{sufnum}(\lambda)\,..\,\mathsf{b}(P) + |P| - 1]$ is a suffix of the palindrome P, and P starts before $\mathsf{sufnum}(\lambda)$. By the definition of P, it is impossible to extend this factor even further.

Maintaining the palindrome P is done as follows: When we advance the variable λ to a leaf with suffix number j ($1 \leq j \leq n$) during the pass, we check whether j is the starting position of a maximal palindrome S. If $|S| > \mathsf{b}(P) + |P| - j$, then S covers more positions of $T[j\,..]$ than P; hence we set P to S. This can be done in constant time by maintaining a pointer to the palindrome in L whose starting position is the smallest among all palindromes whose starting positions are at least j.

Algorithm 8: Computing the non-overlapping reversed LZ77 factorization.

```
1  ST ← suffix tree of T(T[1..n−1])^T#
2  λ^T ← prev_leaf(prev_leaf(smallest_leaf))          ▷ sufnum(λ^T) = 2n − 1
3  λ ← smallest_leaf
4  repeat
5      if λ is corresponding then
6          foreach node v on the path from λ to the root do
7              if v is the root then output fresh factor
8              else if v is marked then                ▷ v is the witness of λ
9                  output length str_depth(v)
10                 output referred position 2n−
                       SA[SA.RMQ[leaf_rank(lmost_leaf(v)),
                                 leaf_rank(rmost_leaf(v))]]
11                 break
12     foreach node v on the path from λ^T to the root do
13         if v is marked then break
14         mark v
15     λ ← next_leaf(λ)
16     λ^T ← prev_leaf(λ^T)
17 until λ = λ^T                                       ▷ it is left to output F_z = $
```

3.7 LZ78 with Space-Efficient Suffix Trees

A natural way to compute the LZ78 factors is to build the LZ trie (recall Sect. 3.3.1) dynamically. The dynamic LZ trie supports the creation of a node, the navigation from a node to one of its children, and the access to the labels of the nodes. The folklore algorithm computing the LZ78 factorization with the dynamic LZ trie works as follows: Given that the LZ trie stores the factors $F_0, F_1, \ldots, F_{x-1}$ (with $F_0 = \Lambda$), the factor F_x is determined by the longest factor F_y (with $y \leq x$) that is a prefix of the suffix $T[1+|F_1 \cdots F_{x-1}| \,..]$, such that F_x is equal to $F_y T[1+|F_1 \cdots F_{x-1}|+|F_y|]$. To find F_y, the algorithm traverses the LZ trie, following an edge from a node to one of its children as many times as F_y is long, which sums up to n. Overall, it searches z times for such a longest factor F_y, and it inserts z times a new leaf into the LZ trie.

However, all (deterministic) dynamic trie implementations have a (log-)logarithmic dependence on σ (or n) for top-down-traversals (recall Fig. 3.1 in the introduction). Our trick is to superimpose the suffix trie (which contains the LZ trie) on the suffix tree such that we can navigate top-down in the LZ trie with level ancestor queries from the suffix *tree* leaves to get rid of this dependence (like we did in Sect. 3.6.1). This superimposition is given in Fig. 3.30 for our

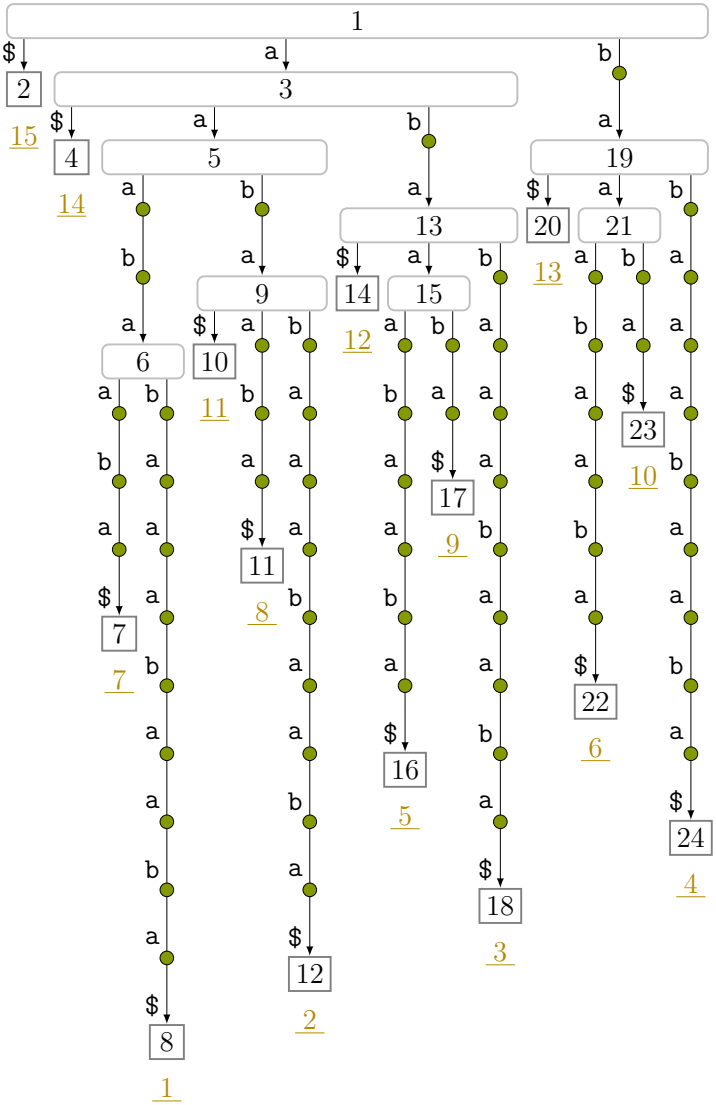


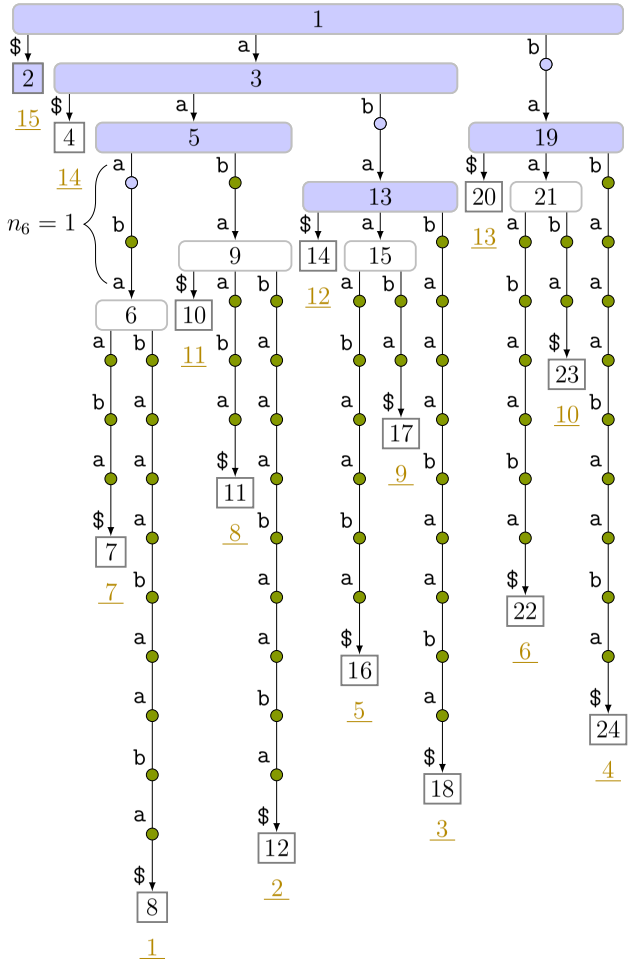
Fig. 3.30: The suffix tree of `aaababaaabaaba$` superimposed by the suffix trie. The suffix trie is obtained (conceptually) by exchanging every suffix tree edge e with $\mathsf{edge\_length}(e) - 1$ new suffix trie nodes. These new suffix trie nodes are represented by the small rounded nodes (●). They represent the *implicit* suffix tree nodes, while the remaining suffix tree nodes represent the *explicit* suffix tree nodes.

running example. How we superimpose the suffix trie (and hence the LZ trie) on the suffix tree is topic of the next section.

3.7.1 Storing the LZ Trie Topology

To ease explanations, we associate a suffix tree edge (u, v) from a node u to a node v uniquely with v such that each suffix tree node v except the root is associated with an edge (u, v).

We borrow the technique from Nakashima et al. [191] to represent the LZ trie with the superimposition of the suffix trie on the suffix tree. In this context, we think of the LZ trie as a connected subgraph of the suffix *trie* containing its root (see Fig. 3.31). Transferred to the suffix *tree*, the LZ (trie) nodes are either already represented by a suffix tree node (*explicit*) or lie



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
SA	15	14	7	1	11	8	2	12	5	9	3	13	6	10	4
B_C	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Preorder	1	3	5	6	9	13	15	19	21
B_V	0	1	1	0	0	1	0	1	0

The horizontal bars groups the entries of B_C into intervals; each interval stores the exploration counter of its respective partially explored node. In each such interval, the rightmost bit is interpreted as the least-significant bit. The edge witness with preorder number 6 is not a witness, since neither its exploration counter was incremented twice nor were any exploration counters of its children incremented.

Fig. 3.31: *Left*: The suffix trie of Fig. 3.30 superimposed by the LZ trie. Blue (■) colored nodes represent the nodes of the LZ trie. *Right*: The bit vectors B_C and B_V as explained at the end of Sect. 3.7.1.

on a suffix tree edge (*implicit*). An implicit LZ node on an edge (u, v) is represented by a node v and its rank within all implicit LZ nodes lying on the same edge (u, v). For our LZ78 factorization algorithm it is important to address the *lowest* LZ nodes on the paths from the suffix tree leaves to the root. Therefore, we keep track of how many implicit LZ nodes are already associated with an edge: For an edge $e = (u, v)$, we define the *exploration counter* n_v with $0 \leq n_v \leq \mathsf{edge\_length}(e)$ storing how far e is *explored*, i.e., how many implicit LZ nodes are associated with e. Adding a factor to the LZ trie results in incrementing one exploration counter. If $n_v = 0$, then the factorization has not (yet) explored e, whereas $n_v = \mathsf{edge\_length}(e)$ tells us that we have already reached v, i.e., v represents an explicit LZ node.

Like for the LZ77 factorization, we mark again certain nodes as witnesses. Here, a witness w will be used for storing the referred indices of factors whose corresponding LZ nodes are on the incoming edge of w. In order to save space, we are interested in a certain type of suffix tree nodes: A *witness* is

- a suffix tree node whose exploration counter is incremented at least *twice* while building the LZ trie, *or*
- a node (whose incoming edge e can have a length $\mathsf{edge\_length}(e) = 1$) having a child whose exploration counter is incremented at least *once* during the parsing.

Algorithm 9: Determining the highest edge (u, v) on the path from the root to λ that is not yet fully explored.

```
   input  : leaf λ
1  function find_edge
2      d ← 0                          ▷ counter for node depth, the depth of the root is zero
3      repeat                         ▷ find first edge on path from root to λ that is not fully explored
4          incr d
5          v ← level_anc(λ, d)
6      until v = λ or B_V[v] = 0
7      u ← parent(v)
8      return (u, v)
```

LZ78 Passes. An LZ78 pass builds the LZ trie topology implicitly by incrementing the exploration counters and marking (in a bit vector B_V) which edges were fully explored. As in Sect. 3.3.4, a pass processes all leaves of the suffix tree in text order. For the LZ78 factorization, we only care about the corresponding leaves. Starting with smallest_leaf, which corresponds to the first factor, we compute the length of the factor corresponding to the currently processed leaf

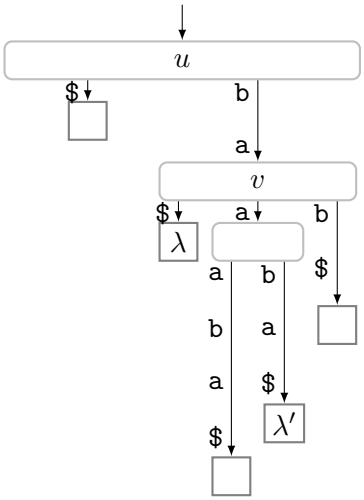


Fig. 3.32: Computing $\mathsf{edge\_length}(u, v)$ with the compressed suffix tree in the setting of Section 3.7.1. The two leaves λ and λ' have v as their LCA whose parent is u. The idea is to compute $\mathsf{edge\_length}(u, v) = \mathsf{str\_depth}(v) - \mathsf{str\_depth}(u)$.

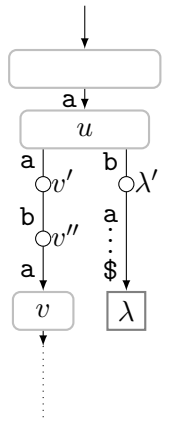
so that we know the distance (in text positions) to the next corresponding leaf. Given a corresponding leaf λ, we want to find the first edge (u, v) from a node u to its child v on the path from the root to λ that is not yet fully explored. The node u is the lowest node on the path that has been marked in B_{V} (cf. Algo. 9). To find u, we traverse the tree downwards along the path from the root to λ by level ancestor queries. After having found u at depth d, we know that its child $v = \mathsf{level\_anc}(\lambda, d + 1)$ is not yet fully explored, i.e., $n_v < \mathsf{edge\_length}(u, v)$. We add a new factor by incrementing n_v by one. If the edge e becomes fully explored (checking this condition is the topic of the next paragraph), we additionally mark v in B_{V}.

Whether the edge e was fully explored or not can be checked as follows: If we use the succinct suffix tree, then we have access to $\mathsf{edge\_length}(e)$, giving us the maximum value of an exploration counter. Otherwise (when using the compressed suffix tree), we first check if v is a leaf, because then the edge (u, v) can be explored at most once due to the following lemma:

Lemma 3.39. Given u be a suffix tree node with a child v, the exploration counter n_v is at most $\min(\mathsf{edge\_length}(u, v), s)$, where s is the number of leaves of the subtree rooted at v.

Proof. Let S be the string of the edge labels on the path from the root to v. Then S occurs exactly s times in T. The exploration counter n_v can only be incremented for a factor having S as a prefix. Therefore, $n_v \leq s$. □

Otherwise (if v is an internal node), we check whether $\mathsf{edge\_length}(u, v) = n_v$ holds, similarly to the computation of $\mathsf{str\_depth}$ as described in Sect. 3.3.3.1: We select a leaf λ' such that v is the LCA of λ and λ' (see Figure 3.32). The idea is that $\mathsf{str\_depth}(v)$ is the length of the LCP of two suffixes corresponding to two leaves having v as their LCA (e.g., λ and λ'). We can compare the m-th character of both suffixes by applying $\mathsf{next\_leaf}$ m times on both leaves before using head. With $m := \mathsf{str\_depth}(u) + n_v + 1$ we can check whether the



	u	v'	λ'	v''	v
u	e	w	w	w	w
v		e	e	w	w
λ			e	e	e

Fig. 3.33: Juxtaposition of witnesses and edge witnesses. Assume that the LZ78 factorization explores the LZ nodes u, v', λ', v'', and v belonging to the subtree (*left*) of the suffix trie in this order (we implicitly map suffix tree nodes to LZ nodes). The *right* tabular figure classifies chronologically the nodes u, v, and λ in witnesses ('w'), edge witnesses ('e'), or plain suffix tree nodes (empty). In each column we create a new LZ78 factor by exploring an LZ node (top row).

edge (u, v) is fully explored. This process also determines the length of the current corresponding factor, which is m. Although we apply **next_leaf** as many times as the factor length, we still get linear time overall, because the lengths of all factors add up to n.

Overall, an LZ78 pass is conducted in $\mathcal{O}(n)$ time, since we query for a level ancestor at most n times. That is because the depth of a suffix tree node is at most its string depth. The number of level ancestor queries is bounded by the sum of the lengths of all factors, which is n.

The suffix tree nodes that we explore during an LZ78 pass are the important nodes for the factorization. The suffix tree node whose exploration counter becomes incremented during a traversal from the root to a leaf λ is called the *edge witness* of λ. A witness is an edge witness, but not necessarily vice versa. Also, while witnesses are always internal nodes, edge witnesses can also be leaves (cf. Fig. 3.33). By definition, the parent of an edge witness is a witness or the root. The set of witnesses with the root and the set of edge witnesses with the root induce a connected subgraph of the suffix tree, respectively.

Bookkeeping the Exploration Counters. Storing the exploration counters in an integer array for all edges costs $2n \lg n$ bits. Our idea is to choose different representations of the exploration counters depending on the state (*not*, *fully*, or *partially explored*). First, there is no need to represent n_v for a node v with parent u until u's incoming edge is fully explored. Second, as the fully explored edges are marked in a bit vector B_{V}, we do not need to store their exploration counters. The third and last type of nodes are those nodes whose parents are fully explored but they are not. In the following, we call them *partially explored nodes*, and show how to maintain their exploration counters in n bits.

The subtrees of all partially explored nodes are disjoint, i.e., each leaf has at most one partially explored ancestor. We associate each partially explored node v with an interval $\mathcal{I}_v = [\mathsf{leaf\_rank}(\mathsf{lmost\_leaf}(v)) \,..\, \mathsf{leaf\_rank}(\mathsf{rmost\_leaf}(v))]$.

These intervals are pairwise disjoint, and the sum of their lengths is at most n. The idea is now to partition a bit vector B_C of length n into these intervals such that we store the exploration counter of a partially explored node v in the space of $B_C[\mathcal{I}_v]$. This space is sufficient since $n_v \leq |\mathcal{I}_v|$ due to Lemma 3.39. By storing n_v in binary coding using the first $\lg |\mathcal{I}_v|$ bits of $B_C[\mathcal{I}_v]$, we can lookup and increment n_v in constant time. After fully exploring the edge of v ($n_v = \mathsf{edge\_length}(u, v)$, where u is v's parent), we clear (the first $\lg |\mathcal{I}_v|$ bits of) $B_C[\mathcal{I}_v]$. As a side effect, this approach resets the counter n_u of every child u of v (the children of v become partially explored on having v fully explored).

Applying this procedure during a pass, we can determine the fully explored edges and maintain n_v of each partially explored node v. The right side of Fig. 3.31 shows B_C after building the LZ trie on our running example, where we stored the exploration counter of the node with preorder number 6 at positions 3 and 4 in B_C.

3.7.2 Alphabet-Sensitive Algorithm

We present two LZ78 factorization variants working with the compressed suffix tree. The first variant is an output-streaming algorithm outputting the coding in text order. For our running example, the output is the list of pairs (`0,a`), (`1,a`), (`0,b`), (`1,b`), (`2,a`), (`3,a`), (`4,a`), (`0,$`), cf. Fig. 3.5.

The other variant builds the LZ trie explicitly in such a way that we can return a factor (its referred index and the character at its end), and perform navigations in the LZ trie in constant time. As an application, we will show that our representation can be enhanced with a data structure (see Lemma 3.44) to support lookups of small substrings of T (which is useful when T is unavailable) in constant time.

Algorithm 10: Pass (a) of the alphabet sensitive LZ78 algorithm of Sect. 3.7.2.

```
1  λ ← smallest_leaf
2  repeat
3  |  (u, v) ← find_edge(λ)
4  |  s ← str_depth(u)
5  |  if v = λ then
6  |  |  λ ← next_leaf^(s+1)(λ)
7  |  |  continue
8  |  B_W[v] ← 1                          ▷ v is an edge witness
9  |  m ← explore(v, λ)                   ▷ m ← n_v; n_v ← n_v + 1
10 |  λ ← next_leaf^(m+s+1)(λ)
11 until λ = smallest_leaf
12 shrink_witness( )                      ▷ see Algo. 12
```

Algorithm 11: Pass (b) of the alphabet sensitive LZ78 algorithm of Sect. 3.7.2.

```
1  B_W.add_rank_support
2  n_v ← 0 for each node v                          ▷ reset exploration counters
3  z_W ← B_W.rank_1(n)
4  W ← array of size z_W lg z bits
5  x ← 1
6  λ ← smallest_leaf
7  repeat
8      (u, v) ← find_edge(λ)
9      s ← str_depth(u)
10     if B_W[v] = 0 then                           ▷ v is not a witness
11         if v is a child of the root then the x-th factor is a fresh factor
12         else the x-th factor refers to W[B_W.rank_1(parent(λ))]
13     else the x-th factor refers to W[B_W.rank_1(v)]
14     incr x
15     if v = λ then
16         λ = next_leaf^(s)[λ]
17         output character head(λ) belonging to the (x − 1)-th factor
18         λ = next_leaf(λ)
19         continue
20     m ← explore(v, λ)                            ▷ m ← n_v; n_v ← n_v + 1
21     λ = next_leaf^(m+s)(λ)
22     output character head(λ) belonging to the (x − 1)-th factor
23     λ = next_leaf(λ)
24 until λ = smallest_leaf
```

3.7.2.1 Output-Streaming Variant

Equipped with the compressed suffix tree of T we do two passes:

(a) create B_W to mark the witnesses (see Algo. 10), and

(b) stream the output by using a helper array mapping witness ranks to factor indices (see Algo. 11).

Pass (a). The goal of this pass is to determine the witnesses. Our idea is to alter the LZ78 pass described in Sect. 3.7.1 in the following way on accessing an edge witness v: If v is an internal suffix tree node whose exploration counter was already incremented, we make v a witness (if it is not yet a witness) by marking v in B_W. If the parent u of v is not the root, we make u a witness if it

Algorithm 12: Filtering the set of edge witnesses to obtain the set of witnesses.

```
   input  : bit vector B_W, exploration counters
1  function shrink_witness
2      foreach node u with B_W[u] = 1 do
3          if n_u ≥ 2 then continue
4          if exists child v of u with n_v ≥ 1 then continue
5          B_W[u] ← 0
```

has not yet been one.[13] A concrete approach is shown in Algo. 10, where we first mark all *edge witnesses* in B_W (Line 8), and then subsequently run Algo. 12 to unmark all edge witnesses in B_W that are no witnesses.

Pass (b). In this pass, we compute the referred indices and the characters at the factor endings in text order. We first focus on the referred indices. We compute them indirectly by assigning each witness to a referred position. Since the witnesses were already found during Pass (a), we can create an array W with $z_W \lg z$ bits to store a factor index for each witness (represented by its witness rank). We fill W in such a way that the referred index of a referencing factor F can be looked up in the entry in W belonging to its witness at the time when processing the leaf corresponding to F. Initially all entries of W are set to $\perp$ (a fixed chosen invalid value).

Before conducting the pass, we reset the exploration counters and B_V. We keep B_W, which we endow with a rank-support to access the witness ranks in constant time.

Assume that we visit the leaf λ corresponding to the x-th factor during the pass, i.e., λ is the x-th visited corresponding leaf. As in Pass (a), we first determine the edge witness v of λ. Then we determine the referred index of the x-th factor by distinguishing two cases (cf. Line 10 in Algo. 11):

- If v is a witness and $y := W[B_W.\mathsf{rank}_1(v)] \neq \perp$, then the x-th factor refers to the y-th factor (the y-th factor is represented by an LZ node that is the parent of the LZ node representing the x-th factor).
- Otherwise (there is no entry in W for v), we have two cases regarding the parent of v:
  - If v is a child of the root, then the x-th factor is a fresh factor.
  - Otherwise (v's parent is a witness), the x-th factor refers to $W[B_W.\mathsf{rank}_1(\mathsf{parent}(v))]$ (the x-th factor is represented by an LZ node that is the first node on the edge from $\mathsf{parent}(v)$ to v).

[13] This can only happen if $\mathsf{edge\_length}(\mathsf{parent}(u), u) = 1$.

Afterwards, if v is a witness, we update W by setting $W[B_{\mathrm{w}}.\mathsf{rank}_1(v)]$ to x.

Up to now, we can output the referred index of the x-th factor during this pass in text order. Finally, the last character of the x-th factor can be obtained by $\mathsf{head}(\lambda')$, where λ' is the leaf that occurs in text order *before* the leaf corresponding to the $(x+1)$-th factor (cf. Lines 17 and 22 in Algo. 11).

We summarize the result of this algorithm in the following theorem:

Theorem 3.40. Given the compressed suffix tree of T, we can compute the LZ78 factorization in $\mathcal{O}(n)$ time using $3n + z \lg z + o(n)$ bits of working space when streaming the output.

Proof. Maintaining the exploration counters as described in Sect. 3.7.1 with the bit vector B_{C} takes n bits. With the space of B_{V} and B_{W} this sums up to $3n + o(n)$ bits of space. The array W uses $z \lg z$ bits. It can be allocated after Pass (a), which determines z. □

Corollary 3.41. We can compute and stream the LZ78 factorization of a text of length n in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space.

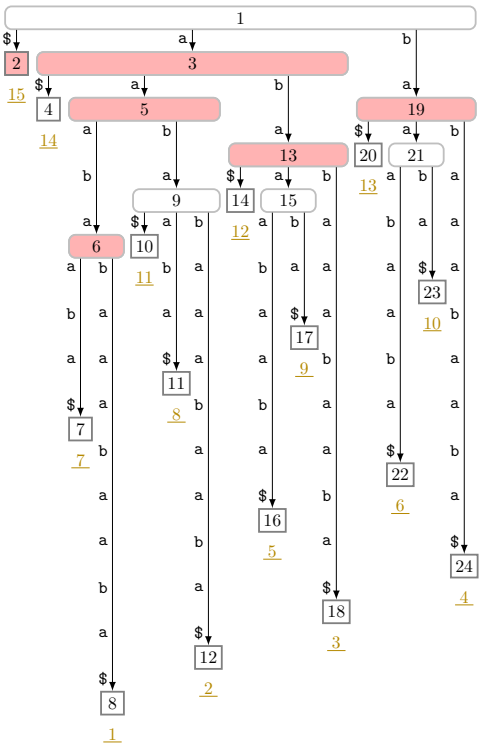
Proof. Analogous to Cor. 3.14. □

3.7.2.2 Explicitly Storing the LZ Trie

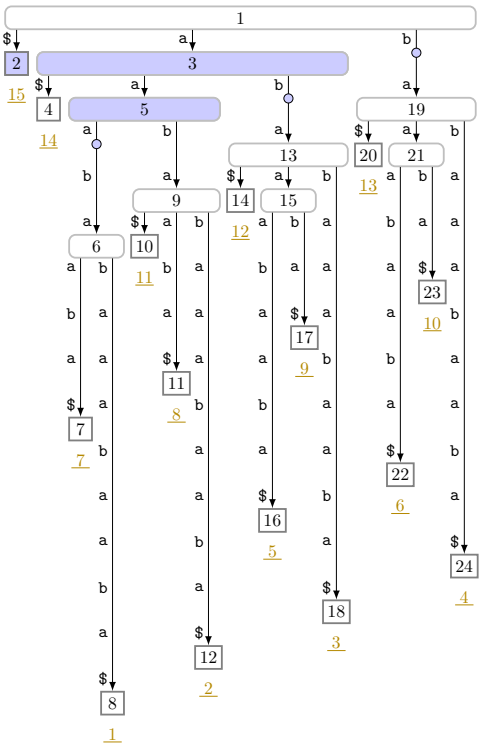
In some applications we are interested in the LZ trie instead of the LZ78 factorization. We provide such an application in Lemma 3.44 that shows a data structure built on top of the LZ trie retrieving small substrings of T efficiently. It is easy to compute the LZ trie after having computed the LZ78 factorization with the approach in Sect. 3.7.2.1. We can modify it to build a pointer-based tree data structure storing the LZ trie in $\mathcal{O}(z \lg z)$ bits. Here, we present a succinct variant of the LZ trie. It is composed of three data structures:

- a BP sequence storing the LZ trie topology,
- an array W' with $z \lg z$ bits storing the factor indices, and
- an array with $z \lg \sigma$ bits storing the last character of each factor, i.e., the labels of the LZ trie edges (see Fig. 3.35).

In this section, we show that the succinct LZ trie can be computed in $\mathcal{O}(n)$ time by altering the approach of Sect. 3.7.2.1. The difference is that (1) we exchange Pass (b) with a pass that computes W' instead of the LZ78 factorization, and that (2) we perform an Euler tour on the suffix tree to compute the other two data structures.



(a) Suffix Tree with Highlighted Edge Witnesses



(b) Superimposed LZ Trie with Highlighted Nodes Marked in B_{LZ}

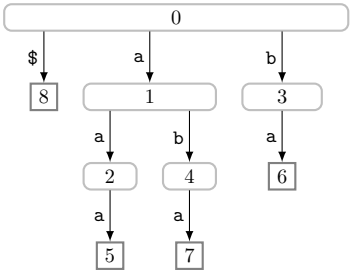
i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
B_{E}	0	1	1	0	1	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0

(c) Bit Vector B_{E}

i	01	234	56	78
BP	(()	((())	(())	(())
B_{LZ}	1	111	10	10

(d) Bit Vector B_{LZ}

Fig. 3.34: The suffix tree (a) and the LZ trie superimposed on the suffix tree (b). The shaded suffix tree nodes in the left image (■) and the shaded LZ nodes in the right image (■) are the ones marked in (c) B_{E} and (d) B_{LZ}, respectively, as explained in Sect. 3.7.2.2. The rows labeled with i are the pre-order numbers of (c) the suffix tree nodes or (d) the LZ nodes, respectively. The bit vectors B_{E} and B_{LZ} mark the same number of nodes. In particular, there is a bijection between the suffix tree nodes marked in B_{E} and the LZ nodes marked in B_{LZ}.



(a) LZ Trie

Preorder Enumeration	01 234 56 78
Balanced Parentheses	(()((())(()))(()))
W'	8 125 47 36
Array of Characters	$ aaa ba ba

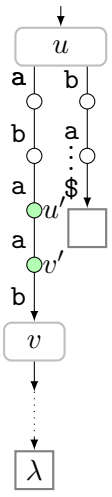
(b) Representation of the LZ Trie in Sect. 3.7.2.2

Fig. 3.35: The LZ trie (a) represented by three data structures (b). The BP sequence represents its topology, the array of factor indices W' stores the labels of the LZ nodes, and the array of characters stores the edge labels (we identify an edge with its incoming node).

In more detail, we build the BP sequence after Pass (a) of Sect. 3.7.2.1. We keep the exploration counters computed during Pass (a) in memory, i.e., we do not clear/erase the exploration counters after the phase has finished. With the exploration counters we can construct the LZ trie since there are exactly n_v LZ nodes on the suffix tree edge of a node v to its parent u. The exploration counter n_v is either

- equal to zero if v's parent u is not fully explored,
- equal to $\mathsf{edge\_length}(u, v)$ if v is fully explored, or
- stored in B_C in case that v is a partially explored node (cf. Sect. 3.7.1).

To build the BP sequence, we perform an Euler tour on the suffix tree, i.e., we traverse the suffix tree with a depth first search starting at the root. We write n_v open (resp. close) parenthesis when visiting a suffix tree node v on walking down (resp. climbing up) during the Euler tour. Although looking up $\mathsf{edge\_length}(u, v)$ for a fully explored node v takes $\mathcal{O}(\mathsf{str\_depth}(v))$ time, this time is bounded by the length of an LZ78 factor whose edge witness is v. An Euler tour can be performed in $\mathcal{O}(n)$ time, since the suffix tree can compute $\mathsf{child}(1)$, $\mathsf{next\_sibling}()$, and $\mathsf{parent}()$ in constant time (recall Sect. 3.3.3).



During the Euler tour we build the $z \lg \sigma$-bits array storing the LZ trie edge labels: Suppose that we create the LZ nodes u' and v' on the suffix tree edge (u, v), where u' is the parent of v'. We can obtain the character of the edge (u', v') by invoking $\mathsf{next\_leaf}$ and head: Given that s is the string depth of u' in the LZ trie, applying $\mathsf{next\_leaf}$ s times to a leaf λ in the subtree rooted at v returns a leaf λ' whose $\mathsf{head}(\lambda')$-value is the label in question. Overall, we can compute the BP representation and the edge labels in $\mathcal{O}(n)$ time taking $2z + z \lg \sigma + o(z)$ bits ($o(z)$ bits for the navigational component [197]).

To compute the array W' storing the factor indices we will exchange Pass (b)

with an alternative LZ78 pass: Suppose that we increment the exploration counter of a node v while processing the x-th factor during this pass. This means that the x-th factor is represented by the n_v-th LZ node on the edge between v and v's parent. Our task is to store the factor index x in W' at the position equal to the preorder number of this LZ node. To compute this preorder number in constant time, we do the following precomputation step, which is the topic of this paragraph: We compute an isomorphic mapping between the edge witnesses and a subset of LZ nodes such that the mapping maps an edge witness v to an LZ node v' with the following properties: First, the LZ trie parent of v' is an explicit LZ node that is represented by the suffix tree parent of v (v has a parent since the root is not an edge witness). The parent of v is a witness. Second, there are $n_v - 1$ LZ nodes below v' forming a (unary) path, where the value of n_v is determined after Pass (a).

To compute this mapping in constant time, we create a bit vector B_{E} of length $2n - 1$ marking suffix tree nodes (including the suffix tree leaves) and a bit vector B_{LZ} of length z marking LZ nodes, see Fig. 3.34. The former marks the edge witnesses, the latter marks the LZ trie children of every explicit LZ node. The last thing we do is adding a rank-support to B_{E}, and a select-support to B_{LZ}.

Now we can map an edge witness v to the LZ node v' with $v' \coloneqq B_{\mathrm{LZ}}.\mathsf{select}_1($ $B_{\mathrm{E}}.\mathsf{rank}_1(v))$ (conceptually $B_{\mathrm{LZ}}.\mathsf{rank}_1(v') = B_{\mathrm{E}}.\mathsf{rank}_1(v)$) such that v' satisfies the above described properties of the mapping.

Finally, we explain the last pass.

Pass (b′). We recompute B_{V} and the exploration counters. We count the current factor index with a variable x. Suppose that we visit the leaf λ corresponding to the x-th factor. Let v be the edge witness of λ. We assign the label x to the LZ node corresponding to the x-th factor by setting $W'[B_{\mathrm{LZ}}.\mathsf{select}_1(B_{\mathrm{E}}.\mathsf{rank}_1(v)) + n_v - 1]$ to x (after incrementing n_v by one like in a standard LZ78 pass). In this operation, we select the LZ node $v' \coloneqq B_{\mathrm{LZ}}.\mathsf{select}_1(B_{\mathrm{E}}.\mathsf{rank}_1(v))$ corresponding to the edge witness v, and select the LZ trie descendant of v' at depth $\mathsf{depth}(v') + n_v - 1$ by adding $n_v - 1$ to the (LZ trie) preorder number of v' (the next $n_v - 1$ descendants of v' are on a unary path below v', thus the preorder numbers of all those descendants are direct successors of the preorder number of v').

The array W' is the last component of our data structure. The following theorem sums up our achieved result:

Theorem 3.42. Given the compressed suffix tree of T, we can build the LZ trie and store it in RAM taking $z(\lg \sigma + \lg z + 2) + o(z)$ bits. The construction algorithm takes $5n + z + o(n)$ additional bits of working space and runs in $\mathcal{O}(n)$ time.

Proof. Maintaining the exploration counters as described in Sect. 3.7.1 takes n bits. With the space of B_{V} (n bits), B_{W} (n bits) and B_{E} ($2n + o(n)$ bits), this

	(a)	(b)	Interim	(M)
X	ISA	L	$L\|W$	Referred Indices
Y	SA	-	-	-

Fig. 3.36: Chronological table of the LZ78 factorization in Sect. 3.7.3, structure equivalent to Fig. 3.13. Having $|X| = n \lg n$ bits and $|Y| = \epsilon n \lg n$ bits, the suffix array stored initially in Y can only be used in conjunction with ISA stored initially in X.

sums up to $5n + o(n)$ bits of space. The bit vector B_{LZ} takes $z + o(z)$ bits. The BP sequence needs $2z + o(z)$ bits. Each of the z LZ nodes stores a factor index using $\lg z$ bits and an ending character using $\lg \sigma$ bits. □

Corollary 3.43. We can compute the LZ trie of a text of length n in $\mathcal{O}(n)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space.

Proof. Analogous to Cor. 3.41. □

Lastly, we introduce an application of our constructed LZ trie sketched in the beginning of this section:

Lemma 3.44 ([219, Lemma 2.6])**.** Given the LZ trie of T, but not necessarily T, there is a data structure built on top of the LZ trie that can recover a substring of T of length $\mathcal{O}(\log_\sigma n)$ in constant time. It takes $z \lg z + 3z \lg \sigma + 5z + \mathcal{O}(n^{3/4} \lg^2 n) + o(z)$ additional bits of space, and can be constructed in-place in linear time.

The data structure of Sadakane and Grossi [219] needs an LZ trie representation that slightly differs from the one that we have computed: It needs an array that maps a factor index to the preorder number of its corresponding LZ node. This array is the inverse of W' (W' is a permutation of integers). In addition, it needs a bit vector B_{T} marking the starting positions of the factors. This bit vector can be computed during Pass (b') by setting $B_{\text{T}}[\textsf{sufnum}(\lambda)]$ to 1 for each corresponding leaf.

Corollary 3.45. Given a text T, the data structure of Lemma 3.44 can be computed in linear time with $\mathcal{O}(n \lg \sigma)$ bits of working space. It takes $2z \lg z + 4z \lg \sigma + 7z + \mathcal{O}(n^{3/4} \lg^2 n) + o(z)$ bits of space in total (including the LZ trie).

3.7.3 Alphabet-Independent Algorithm

In this variant we store the referred indices in the array X and mark the starting positions of the factors in a bit vector B_{T} of length n such that the referred index

Algorithm 13: Pass (b) of the alphabet independent LZ78 algorithm of Sect. 3.7.3.

result : $L[x]$ = edge witness of the x-th factor

$\lambda \leftarrow$ smallest_leaf
$L \leftarrow X[1\,..\,z]$ ▷ L is a pointer to $\mathsf{ISA}[1\,..\,z]$
repeat
  $(u, v) \leftarrow$ find_edge(λ) ▷ $u \in B_{\mathrm{V}}, v \notin B_{\mathrm{V}}$
  $L[\text{sufnum}(\lambda)] \leftarrow v$
  $s \leftarrow$ str_depth(u)
  if $v = \lambda$ then
    $\lambda = \text{next\_leaf}^{(s+1)}(\lambda)$
    $B_{\mathrm{T}}[\text{sufnum}(\lambda) + s + 1] \leftarrow 1$
    continue
  $B_{\mathrm{W}}[v] \leftarrow 1$ ▷ v is an edge witness
  $m \leftarrow \text{explore}(v, \lambda)$ ▷ $m \leftarrow n_v; n_v \leftarrow n_v + 1$
  $B_{\mathrm{T}}[\text{sufnum}(\lambda) + s + 1 + m] \leftarrow 1$
  $\lambda = \text{next\_leaf}^{(m+s+1)}(\lambda)$
until $\lambda =$ smallest_leaf
shrink_witness( ) ▷ see Algo. 12

Algorithm 14: Matching (M) in the alphabet independent LZ78 algorithm of Sect. 3.7.3.

result : $L[x]$ = referred index of the x-th factor

B_{W}.add_rank_support
$z_{\mathrm{W}} \leftarrow B_{\mathrm{W}}.\text{rank}_1(n)$
$W \leftarrow X[z + 1\,..\,n]$ ▷ L is a pointer to $\mathsf{ISA}[1\,..\,z]$
for $1 \le x \le z$ do
  $v \leftarrow L[x]$ ▷ v is the edge witness of the x-th factor
  if $B_{\mathrm{W}}[v] = 0$ or $W[B_{\mathrm{W}}.\text{rank}_1(v)] = 0$ then
    if v is a child of the root then
      $L[x] \leftarrow 0$ ▷ x-th factor is a fresh factor
    else $L[x] \leftarrow W[B_{\mathrm{W}}.\text{rank}_1(\text{parent}(v))]$
  else $L[x] \leftarrow W[B_{\mathrm{W}}.\text{rank}_1(v)]$
  if $B_{\mathrm{W}}[v] = 1$ then $W[B_{\mathrm{W}}.\text{rank}_1(v)] \leftarrow x$

and the last character of the x-th factor are $X[x]$ and $T[B_{\mathrm{T}}.\mathsf{select}_1(x+1)-1]$, respectively.

Having $\mathcal{O}(n)$ bits of working space on top of the space used by the succinct suffix tree, our idea is to overwrite the array X to free up space. We will overwrite X multiple times during the factorization (see Fig. 3.36). After overwriting X (storing ISA initially), we will no longer have access to SA and therefore cannot evaluate $\mathsf{edge\_length}$ needed to decide whether an edge is fully explored. Fortunately, it is sufficient to know the maximum exploration counter of every witness — a node that is not a witness has an exploration counter of either zero (not touched during an LZ78 pass) or one (represented as an LZ78 trie leaf). We therefore aim at storing the maximum value of the exploration counter of every witness before overwriting X. The maximum values can be determined in one pass. We can store them in a bit vector B_{L} of length at most $z+z_{\mathrm{W}}$, since we increment the exploration counters exactly z times. Technically speaking, we store the exploration value n_w unary with $\mathsf{0}^{n_w}\mathsf{1}$ in B_{L} for every witness w; hence B_{L} has z_{W}-many ones. We store the values sorted by the preorder numbers of the witnesses such that $B_{\mathrm{L}}.\mathsf{rank}_1(B_{\mathrm{W}}.\mathsf{rank}_1(w))-B_{\mathrm{L}}.\mathsf{rank}_1(B_{\mathrm{W}}.\mathsf{rank}_1(w)-1)-1$ returns n_w (we stipulate that $\mathsf{rank}_1(0) \coloneqq 0$).

The algorithm is divided into two passes and a matching phase:

(a) Determine the witnesses and mark them in B_{W}. Write the maximum exploration counters of each *witness* in B_{L} unary.

(b) Mark the starting positions of the factors in a bit vector B_{T}, and create an array $L[1\,..\,z]$ storing the preorder number of the *edge witness* of each factor (see Algo. 13).

(M) Use the witness ranks (a subset of the ranks of the edge witnesses) stored in the array $L[1\,..\,z]$ to identify the referred indices (see Algo. 14).

Pass (a). After performing a single LZ78 pass we have determined the witnesses and the edge witnesses (this can be done as in Algo. 10). The goal of this pass is to create the bit vector B_{L} by determining the exploration counter of every witness. Due to our bookkeeping method of the exploration counters in Sect. 3.7.1, we know the exploration counter of all partially explored edges. Further, the maximum exploration counter of a fully explored node v is equal to the value of $\mathsf{edge\_length}(u,v)$, where (u,v) is the edge between the node v and its parent u. Altogether we have the information needed to create the bit vector B_{L}. The exploration values stored in B_{L} are computed by an Euler tour on the suffix tree in depth first order: If a witness w is marked in B_{V}, then n_w is equal to $\mathsf{edge\_length}(u,w)$, where u is the parent of w. Otherwise (w is not marked in B_{V}), n_w is stored in the bit vector B_{C} ($n_w > 0$ since w is a witness).

With B_{L} we no longer need access to SA (e.g., for accessing $\mathsf{edge\_length}$). This is crucial, since we can access SA only when the arrays X and Y are left

z	#internal-nodes	#leaves
z_{F}	α	β
z_{R}	γ	δ

Fig. 3.37: Contingency table depicting the partitioning of all factors in the proof of Lemma 3.46. The columns #internal-nodes and #leaves are defined as the number of internal nodes and the number of leaves in the LZ trie, respectively.

untouched (X stores ISA and Y stores the data structure of Lemma 3.7). In the next pass, we will overwrite the ISA entries stored in X after they are no longer used.

Pass (b). The main goal of this pass is the creation of the array L. We create its contents sequentially during the pass: When performing a traversal from a leaf λ corresponding to the x-th factor, we write v to $L[x]$, where v is the *edge* witness of λ (cf. Line 5 of Algo. 13). With L it is easy to find the referred index y of a referencing factor F_x. That is because either (a) F_y shares the edge witness with F_x, or (b) $L[y]$ is the parent node of $L[x]$. The latter condition always holds if $n_{L[x]} = 1$, in particular if $L[x]$ is a leaf because we can create at most one LZ78 node on the edge to a leaf (recall Lemma 3.39).

Since L takes $z \lg n$ bits, we can store L in the first z positions of the array X. Although X stores ISA, necessary for next_leaf, we will visit the same leaf never again such that we can sequentially overwrite $X[x]$ with $L[x]$ for all $1 \leq x \leq z$ in increasing order.

We can compute B_{T} simultaneously: Given the leaf λ corresponding to a factor F, the length of F is computed by summing up the lengths of the edge labels on the traversed path from the root to its edge witness v (i.e., the length of the string label of v), and adding n_v's value.

Matching (M). Matching the factors with their references can now be done with L in a straightforward manner. Let us consider a referencing factor F_x having the referred factor F_y. We have two cases: Whenever F_y is explicitly represented by a node v (i.e., by F_y's witness), v is the parent of F_x's witness. Otherwise, F_y has an implicit representation and hence has the same witness as F_x: If $L[x]$ does not occur in $L[1\,..\,x-1]$, then F_y is determined by the largest position y with $y < x$ and $L[y] = \mathsf{parent}(L[x])$; otherwise ($L[x]$ is *not* the first occurrence of $L[x]$ in L), the referred factor of F_x is determined by the largest y with $y < x$ and $L[x] = L[y]$.

Remember that we store L in $X[1\,..\,z]$, leaving us $X[z+1\,..\,n]$ as free working space that will be occupied by a new array W, storing for each witness w the index of the most recently processed factor whose witness is w. Fortunately, this space is sufficient due to the following lemma:

Lemma 3.46. $z + z_{\mathrm{W}} \leq n$.

Proof. Let α and β be the number of fresh factors that are internal LZ nodes and LZ trie leaves, respectively. Also, let γ and δ be the number of referencing factors that are internal LZ nodes and LZ trie leaves, respectively. Obviously, $\alpha + \beta + \gamma + \delta = z$ (see Fig. 3.37). With respect to the factor length, each referencing factor has a length at least 2, while each fresh factor is exactly one character long. Hence $n \geq 2(\gamma + \delta) + \alpha + \beta = z + \gamma + \delta$. Since each LZ trie leaf that is counted by δ has an internal LZ node of depth one as ancestor (counted by α), $\alpha \leq \delta$ holds. Since the number of witnesses is bounded by the number of internal LZ nodes, we obtain $z + z_{\mathrm{W}} \leq z + \alpha + \gamma \leq z + \gamma + \delta \leq n$. □

Finally, we describe how to convert L (stored in $X[1 \mathinner{..} z]$) to the referred indices, such that in the end $X[x]$ contains the referred index of F_x for $1 \leq x \leq z$. We scan $L = X[1 \mathinner{..} z]$ from left to right (cf. Algo. 14). During the scan, for each witness v, we keep track of the index of the most recently visited factor F whose witness is v by storing F's index in $W[B_{\mathrm{W}}.\mathsf{rank}_1(v)]$.

Suppose that we process F_x with $v \coloneqq L[x]$ (cf. Line 5 in Algo. 14).

- If v is not a witness or $W[B_{\mathrm{W}}.\mathsf{rank}_1(v)]$ is empty, we check the parent of v. If v is a child of the root, then F_x is a fresh factor. Otherwise, its referred index is $W[B_{\mathrm{W}}.\mathsf{rank}_1(\mathsf{parent}(v))]$.
- Otherwise (v is a witness and $W[B_{\mathrm{W}}.\mathsf{rank}_1(v)]$ has a value), the referred index of F_x is $W[B_{\mathrm{W}}.\mathsf{rank}_1(v)]$.

In either case, if v is a witness, we update W by writing the current factor index x to $W[B_{\mathrm{W}}.\mathsf{rank}_1(v)]$. After processing F_x, we no longer need the value $X[x]$. Hence, we can write the referred index of F_x to $X[x]$ (if it is a referencing factor) or set $X[x]$ to 0 (if it is a fresh factor). In the end, $X[1 \mathinner{..} z]$ stores the referred indices of the referencing factors. Overall we obtain the following result:

Theorem 3.47. Allowing the succinct suffix tree of T to be *rewritable*, we can overwrite it with the LZ78 factorization in $\mathcal{O}(n)$ time using $3n + o(n)$ bits of working space.

Proof. Maintaining the exploration counters as described in Sect. 3.7.1 with B_{C} takes n bits. With the space of B_{T} marking the starting positions of the factors and B_{L} storing the maximum exploration values of the witnesses, this sums up to $3n + o(n)$ bits of space. □

Corollary 3.48. We can compute the LZ78 factorization of a text of length n in $\mathcal{O}(n/\epsilon)$ time using $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of space. The factors are stored in-place.

Proof. We create the succinct suffix tree of Thm. 3.9 to compute the LZ78 factorization. During the factorization, we overwrite the space of X (used for representing ISA) to compute the LZ78 factorization in-place according to Thm. 3.47. □

Note that the representation of the LZ78 factorization using $\mathcal{O}(n)$ bits still needs the text to be present for accessing the last character of a factor (see Sect. 3.3.4).

3.8 Practical LZ78 and LZW Computation

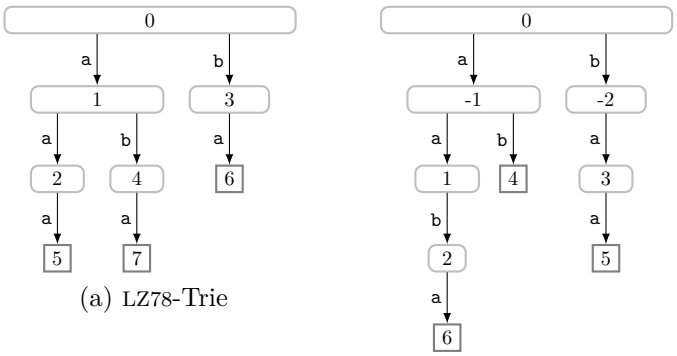
Although we proposed two linear time algorithms computing the LZ78 factorization in Sect. 3.7, none of them will become practical without having a practical suffix tree implementation available. Instead of constructing the suffix tree, practical LZ78 factorization algorithms maintain the LZ trie with a dynamic trie data structure (see the beginning of Sect. 3.7). We follow this approach, and present a thorough study on dynamic trie data structures subject to computing the LZ78 factorization. Within this section, we drop the requirement of the input text T to end with the delimiter `$`. To emphasize on that, we carry on our running example without the delimiter `$` such that our running example of this section becomes $T =$ `aaababaaabaaba`.

Additionally to the LZ78 factorization, we shed a light on the LZW factorization. If we stipulate that F_0 and $F_{z+1}[1]$ are the empty string, we can formulate the definition of the LZW factorization as follows:

Definition 3.49 ([242])**.** A factorization $F_1 \cdots F_z = T$ is called the *LZW factorization* [242] of T if, for all $1 \leq x \leq z$, $F_x = F_y F_{y+1}[1]$ with $F_y = \operatorname{argmax}_{S \in \{F_{y'} | 1 \leq y' \leq x-1\}} |S|$, or $F_x = c \in \Sigma$ if no such F_y exists. If $F_x = F_y F_{y+1}[1]$ for an integer y with $1 \leq y \leq x-1$, we call y the *referred index* of the factor F_x. Otherwise, $F_x = c$ for a $c \in \Sigma$; we set its referred index to $-c < 0$.

We transform the list of LZW factors to a list of integer values as follows: We linearly process each factor F_x for $1 \leq x \leq z$. If F_x's referred index is not positive, F_x is equal to a character c, and we output the value $-c$. Given a factor $F_x = F_y F_{y+1}[1]$ with a referred index $y > 0$, we output y. The LZW factorization of $T =$ `aaababaaabaaba` as defined in Def. 3.49 is $T =$ $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aa}}|\overset{3}{\texttt{b}}|\overset{4}{\texttt{a}}|\overset{5}{\texttt{ba}}|\overset{6}{\texttt{aab}}|\overset{7}{\texttt{aaba}}$. We output it as `-1|1|-2|-1|3|2|6`. Remembering Def. 3.3, the LZ78 factorization of T is $T =$ $\overset{1}{\texttt{a}}|\overset{2}{\texttt{aa}}|\overset{3}{\texttt{b}}|\overset{4}{\texttt{ab}}|\overset{5}{\texttt{aaa}}|\overset{6}{\texttt{ba}}|\overset{7}{\texttt{aba}}$, where the vertical bars separate the factors. The LZ78 factorization is output as `a|(1,a)|b|(1,b)|(2,a)|(3,a)|(4,a)`. We observe that each LZW factor is coded by a single integer, whereas the LZ78 coding produces tuples for the referencing factors.

Like the LZ78 factorization, the LZW factors can be represented in an LZ trie. Each LZW factor F_x, except the last one, is represented by a trie node v labeled with x ($1 \leq x \leq z-1$) such that the parent u of v is labeled with y if y is the referred index of F_x. The edge (u, v) is then labeled with the first character of F_{x+1}. A juxtaposition of the LZ78 and LZW tries built on $T =$ `aaababaaabaaba` is given in Fig. 3.38. The focus of the rest of this section is on analyzing different trie representations. All representations are subject to the LZ78 or LZW



(a) LZ78-Trie (b) LZW-Trie

Fig. 3.38: LZ78 trie (a) and LZW trie (b) of $T = \texttt{aaababaaabaaba}$.

factorization, which we compute with the algorithm described at the beginning of Sect. 3.7.

3.8.1 LZ-Trie Representations

We present five representations, each providing different trade-offs for speed and memory consumption. All representations have in common that they work with dynamic arrays. When working with dynamic arrays, one has to think about how to resize them efficiently.

Resize hints. The usual strategy for dynamic arrays is to double the size of an array when it gets full. To reduce the memory consumption, a hint on how large the number of factors z might become is advantageous to know for a dynamic LZ trie data structure. We provide such a hint based on the following lemma:

Lemma 3.50 ([74, Sect. C],[20, Lemma 1])**.** The number of LZ78 factors z is at least $\sqrt{2n + 1/4} - 1/2$.

At the beginning of the factorization, we let a dynamic trie reserve enough space to store at least $\sqrt{2n}$ elements without resizing. On enlarging a dynamic trie, we usually double its size. However, if the number of *remaining characters* r to parse is below a certain threshold, we try to scale the data structure up to a value for which we expect that all factors can be stored without resizing the data structure again. Let z' be the currently computed number of factors. If $r > n/2$ we use $z' + 3r/\lg r$ as an estimate (the number 3 is chosen empirically[14]), derived from $z - z' = \mathcal{O}(r/\log_\sigma r)$ based on Lemma 3.4, otherwise we use $z' + z'r/(n-r)$ derived from the expectation that the ratio between z' and $n-r$ will be roughly the same as between z and n (interpolation).

[14] There are artificial texts like $\texttt{a}^n$ for which we overestimate the number of factors.

i	1	2	3	4	5	6	7
First Child	2	5	6	7			
Next Sibling	3	4					
Character	a	a	b	b	a	a	a

Fig. 3.39: Array data structures of binary storing the LZ78 factorization of the example given in Fig. 3.38.

3.8.1.1 Deterministic LZ Tries

We first recall two trie implementations using arrays to store the node labeled with x at position x, for each x with $1 \leq x \leq z$.

Binary search trie. The first-child next-sibling representation binary maintains its nodes in three arrays. A node stores a pointer to one of its children, and a pointer to one of its siblings. It additionally stores the label (i.e., a character) of the edge to its parent. The trie binary takes $2z \lg z + z \lg \sigma$ bits when storing z nodes. We do not sort the nodes in the trie according to the character on their incoming edge, but store them in the order in which they are inserted.[15] Figure 3.39 gives an example. To navigate from a node v to its child with label $c \in \Sigma$, we take the first child of v and then sequentially scan all its next siblings until finding a node storing the character c.

Ternary search trie. The ternary search trie [30] ternary differs from binary in that a ternary node stores one more pointer to a sibling: A node of ternary stores a character, a pointer to one of its children, a pointer to one of its smaller siblings, and a pointer to one of its larger siblings. The trie ternary takes $3z \lg z + z \lg \sigma$ bits when storing z nodes. Similar to binary, we do not rearrange the nodes. To navigate from a node v to its child with label $c \in \Sigma$, we take the pointer to one of its children and then binary search for the sibling storing the character c (given that we are at a node storing a character d, we take its smaller sibling if $c < d$, otherwise its larger sibling).

3.8.1.2 LZ Tries with Hashing

We use a hash table $H[0..M-1]$ for a natural number M, and a hash function h to store key-value pairs. We determine the position of a pair (k, v) in H by the *initial address* $h(k) \mod M$; we handle collisions with linear probing. We enlarge H when the maximum number of entries $m := \alpha M$ is reached, where α is a real number with $0 < \alpha < 1$.

A hash table can simulate a trie as follows: Given a trie edge (u, v) with label c, we use the unique key $c + \sigma\ell$ to store v, where ℓ is the label (factor index) of u (the root is assigned the label 0). This allows us to find and create

[15] We found this faster in our experiments.

nodes in the trie by simulating top-down-traversals. This trie implementation is called hash in the following.

Table size. We choose the hash table size M to be a power of two. Having $M = 2^k$ for $k \in \mathbb{N}$, we can compute the remainder of the division of a hash value by the hash table size with a bitwise AND operation, i.e., $h(x) \mod 2^k = h(x)$ & $(2^k - 1)$, which is practically faster[16].

If the aforementioned resize hint suggests that the next power of two is sufficient for storing all factors, we set $\alpha \leftarrow 0.95$ (which turned out to be a good value in the experiments) before enlarging the size (if necessary). We also implemented a hash table variant that changes its size to fit the provided hint. This variant then cannot use the fast bit mask to simulate the operation mod M. Instead, it uses a practical alternative that scales the hash value $h(k)$ of a key k by M and divides this value by the largest possible hash value[17], i.e., $Mh(k)/(\max_{k'} h(k'))$. We mark those hash table variants with a plus sign, e.g., hash+ is the respective variant of hash.

Reasons for linear probing. Linear probing inserts a tuple with key k at the first free entry, starting at the initial address. Probing for the free entry closest to the initial address is cache-efficient since the probing scans *linearly* the entries starting at the initial address. Using large hash tables and small keys, the cache-efficiency can compensate the chance of higher collisions [17, 123]. Linear probing excels if the load ratio is below 50%, and it is still competitive up to a load ratio of 80% [34, 180]. Nevertheless, its main drawback is *clustering*: Linear probing creates runs, i.e., entries whose hash values are equal. With a sufficient high load, it is likely that runs can merge such that long sequence of entries with different hash values emerge. When trying to look up a key k, we have to search the sequence of succeeding elements starting at the initial address until finding a tuple whose key is k, or ending at an empty entry. Fortunately, the expected time of a search is rather promising for an α not too close to one: Given that the used hash function h distributes the keys independently and uniformly, we get $\mathcal{O}(1/(1-\alpha)^2)$ expected time for a search [154, Sect. 6.4]. In practice, even weak hash functions (like the ones we use in this section) tend to behave as truly independent hash functions [48]. These properties convinced us that linear probing is a good candidate for our representations of the LZ trie using a hash table.

[16] http://blog.teamleadnet.com/2012/07/faster-division-and-modulo-operation.html

[17] http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-hash-table-with-tiny-memory-footprints/

3.8.1.3 Compact Hashing

In terms of memory, hash is at a disadvantage compared to binary, because the key-value pairs consist of two factor indices and a character; for an $\alpha < 1$, hash always takes more space than binary. To reduce the size of the stored keys, we introduce the representation cht using compact hashing.

The idea of compact hashing [77, 154] is to use a *bijective* hash function such that when storing a tuple with key k in H, we only store the value and the quotient $\lfloor h(k)/M \rfloor$ in the hash table. The original key of an entry of H can be restored by knowing the initial address $h(k) \mod M$ and the stored quotient $\lfloor h(k)/M \rfloor$. To address collisions and therefore the displacement of a stored entry due to linear probing, Cleary [50] adds two bit vectors, each of length M, with which the initial address can be restored. (One bit vector marks the initial addresses of all stored elements, and the other marks the boundaries of groups of elements having the same initial address.)

For the bijective hash function h, we consider two classes:

Linear Congruential Generators The class of linear congruential generators (LCGs) [42] contains all functions $\mathsf{lcg}_{a,b,p} : [0..p-1] \to [0..p-1], x \mapsto (ax+b) \mod p$ with $p \in \mathbb{N}, 0 < a < p, 0 \leq b < p$. If p and a are relative prime ($\gcd(p,a) = 1$), then there exists a unique modular multiplicative inverse $a^{-1} \in [1..p-1]$ of a such that $aa^{-1} \mod p = 1$. Then $\mathsf{lcg}^{-1}_{a,b,p} : y \mapsto (y-b)a^{-1} \mod p$ is the inverse of $\mathsf{lcg}_{a,b,p}$. If p is prime, then $a^{-1} = a^{p-2} \mod p$ due to Fermat's little theorem. Otherwise, the extended Euclidean algorithm can compute two numbers a^{-1} and y such that $aa^{-1} + py = \gcd(p,a) = 1$, i.e., $aa^{-1} = 1 \mod p$.

Xorshift Functions The xorshift hash function class [185] contains functions that use shift and exclusive OR (XOR) operations. Let $\oplus$ denote the binary XOR operator and w the number of bits of the input integer. For an integer $j < -\lfloor w/2 \rfloor$ or $j > \lfloor w/2 \rfloor$, the xorshift operation $\mathsf{sxor}_{w,j} : [0..2^w-1] \to [0..2^w-1], x \mapsto (x \oplus (\lfloor 2^j x \rfloor \mod 2^w)) \mod 2^w$ is inverse to itself: $\mathsf{sxor}_{w,j} \circ \mathsf{sxor}_{w,j} = \mathsf{id}$.

It is possible to create a bijective function that is a concatenation of functions of both families[18].

A compact hash table can use less space than a traditional hash table if the size of the keys is large: If the largest integer key is u, then all keys can be stored in $\lceil \lg u \rceil$ bits, whereas all quotients can be stored in $\lceil \lg(\max_u h(u)/M) \rceil$ bits. By choosing a hash function h with $M \leq \max_u h(u) \leq cM$ for a constant $c > 1$, it is possible to store the quotients in a number of bits independent of the number of the keys.

[18] Popular hash functions like MurmurHash 3 (https://github.com/aappleby/smhasher) use a post-processing step that applies multiple LCGs $\mathsf{lcg}_{a,0,2^{64}}$ with a as a predefined odd constant, and some xorshift-operations.

Enlarging the hash table. On enlarging the hash table, we choose a new hash function, and rebuild the entire table with the new size and a newly chosen hash function. We first choose a hash function h out of the aforementioned bijective hash classes and adjust h's parameters such that h maps from $[0\,..\,2m\sigma-1]$ to $[0\,..\,2m\sigma-1]$ (m already has its new size). This means that

- we select a function $\mathsf{lcg}_{a,b,p}$ with a prime $m\sigma < p < 2m\sigma$ (such a prime exists [233] and can be precomputed for all $M = 2^k$, $1 \le k \le \lg n$) and $0 < a, b \le p$ randomly chosen, or that
- we select a function $\mathsf{sxor}_{w,j}$ with $\lg(m\sigma) \le w \le \lg(2m\sigma)$ and j arbitrary.

Note that although the domain of h is $[0\,..\,2m\sigma-1]$, we apply h only to keys belonging to $[0\,..\,m\sigma-1]$.

The hash table always stores trie nodes with labels that are at most m; this is an invariant due to the following fact: before inserting a node with label $m+1$ we enlarge the hash table and hence update m. Therefore, the key of a node can be represented by a $\lceil \lg(m\sigma) \rceil$-bit integer (we map the key to a single integer with $[0\,..m-1]\times[0\,..\sigma-1] \to [0\,..m\sigma-1], (y,c) \mapsto (\sigma y+c)$. Since h is a bijection, the function $[0\,..m\sigma-1] \to [0\,..M-1]\times[0\,..\,\lfloor(2m\sigma-1)/M\rfloor], i \mapsto (h_1(i), h_2(i)) \coloneqq (h(i) \mod M, \lfloor h(i)/M \rfloor)$ is injective. The value of h_1 determines the initial address of an entry. When we want to store a node with label x and key $y\sigma+c$ in the hash table, we put x and $h_2(\sigma y+c)$ in an entry of the hash table. The entry is determined by h_1, the linear probing strategy, and a re-arrangement with the bit vectors. It stores x using $\lg m$ bits and $h_2(\sigma y+c)$ using $\lg(2\alpha\sigma)$ bits. In total, we need $M(\lg(2\alpha\sigma)+\lg m)+2M$ bits to store m elements in a compact hash table of size M. Since $m \le 2z-1$, there is a power of two such that $M = 2^{\lfloor \lg(z/\alpha)\rfloor+1} \le (2z-1)/\alpha$. On termination, the compact hash table takes at most $M(2+\lg(2\alpha\sigma m)) \le (2z-1)(3+\lg(\alpha\sigma z))/\alpha$ bits.

Lemma 3.51. There is a randomized algorithm computing the LZ78 and LZW factorization online with at most $z(3\lg(z\sigma\alpha)+11)/\alpha$ bits of working space, for a fixed α with $0 < \alpha < 1$ and linear expected running time.

Proof. The memory peak is reached when we have to copy the data from the penultimate table to the final hash table with the above size. It is at most $M(3+\lg(m\alpha\sigma)) + (M/2)(2+\lg(m\alpha\sigma)) \le (2z-1)(11+3\lg(z\alpha\sigma))/2\alpha$. □

Comparing the memory peak of cht with the approach using a classic hash table (where we need to store the full key), we see that hash has a memory peak of $M(\lg m + \lg m + \lg\sigma) + (M/2)(\lg(m/2)+\lg(m/2)+\lg\sigma) \le 3(2z-1)(4/3+\lg(\sigma z^2))/\alpha$ bits.

Theorem 3.52. We can compute the LZ78 and LZW factorization online with a Las Vegas algorithm using $\mathcal{O}(n/\epsilon)$ time and $\mathcal{O}((1+\epsilon)z\lg(\sigma z))$ bits of working space, for a constant ϵ with $0 < \epsilon \le 1$.

Trie	Space Best Case	Space Worst Case
binary	$3z(\lg(z^2\sigma) - 2/3)/2$	$3z(\lg(z^2\sigma) + 4/3)$
ternary	$3z(\lg(z^3\sigma) - 1)/2$	$3z(\lg(z^3\sigma) + 2)$
hash	$3z(\lg(z^2\sigma) - 2/3)/2\alpha$	$6z(\lg(z^2\sigma) + 4/3)/\alpha$
cht	$3z(\lg(\alpha z\sigma) + 8/3)/2\alpha$	$3z(\lg(\alpha z\sigma) + 11/3)/\alpha$
rolling	$3z(w + \lg(z) - 1/3)/2\alpha$	$6z(w + \lg(z) + 2/3)/\alpha$

Fig. 3.40: Upper and lower bounds of the maximum memory (measured in bits) used during an LZ78/LZW factorization with z factors. The size of a fingerprint is w bits.

For the evaluation, we use a preliminary version of the implementation of Poyias et al. [207] that is based on [50] with the difference that Cleary uses bidirectional probing ([207] uses linear probing).

3.8.1.4 Rolling Hashing

Here, we present an alternative trie representation with hashing, called rolling. The idea is to maintain the Karp-Rabin fingerprints [149] of all computed factors in a hash table such that the navigation in the trie is simulated by matching the fingerprint of a substring of the text with the fingerprints in the hash table. Given that the fingerprint of the substring $T[i\,..\,i+\ell-1]$ matches the fingerprint of the string read on the path from the LZ trie root to a node u, we can compute the fingerprint of $T[i\,..\,i+\ell]$ to find the child of u that is connected to u by an edge with label $T[i+\ell]$ (with high probability). To compute the fingerprints, we choose one of the two rolling hash function families:

- The function $\mathsf{ID37}(T) = \sum_{i=1}^{|T|} h(T[i])37^{|T|-i} \mod 2^w$, where w is the word size and h is a hash function that maps the alphabet uniformly to the range $[0\,..\,2^{32}-1]$. It belongs to the randomized Karp-Rabin ID37 family [172][19].
- The function $\mathsf{fermat}(T) = \sum_{i=1}^{|T|}(T[i]-1)(\sigma+1)^{|T|-i} \mod 2^w$, where w is the word size. The modulo by the maximum value 2^w that fits into a word surrogates the integer overflow. The value $T[i]-1$ is in the range $[0\,..\,\sigma-1]$. In the case of a byte alphabet, $\sigma+1 = 2^8+1 = 257$ is a Fermat prime [212]. We compute $\mathsf{fermat}(T)$ with Horner's rule.

Both rolling hash functions discard the classic modulo operation with a prime number in favor of integer overflows due to performance reasons; this trick was already suggested in [113]. The LZ78/LZW computation using rolling is a Monte Carlo algorithm, since the computation can produce a wrong factorization if the computed fingerprints of two different strings are the same (because the fingerprints *are* the hash table keys).

[19] https://github.com/lemire/rollinghashcpp

Compressors	
Coder wrapper	`encode`
BWT [40]	`bwt`
LCPComp [69, Sect. 3.2]	`lcpcomp`
LZSS (Def. 3.1)	`lzss_lcp`
LZSS with CST (Sect. 3.4.2)	`lzsscics`
LZW [242]	`lzw`
LZ78 (Def. 3.3)	`lz78`
LZ78 with CST (Sect. 3.7.2)	`lz78cics`
Move-To-Front	`mtf`
Run-Length-Encoding	`rle`

Coders	
Bit-Compact	`bit`
Elias-γ [73]	`gamma`
Elias-δ [73]	`delta`
Huffman [243, Sect. 2.3]	`huff`

Fig. 3.41: A selection of compressors and the coders implemented in tudocomp. Each compressor and coder receives an identifier (right column of each table). The output of a compressor can be processed by a coder to produce a smaller compressed file.

3.8.1.5 Summary of All Dynamic Trie Data Structures

We summarize the description of the trie data structures in this and the previous section by Fig. 3.40 showing the maximum space consumption of each described trie. The maximum memory consumption is due to the peak at the last enlargement of the dynamic trie data structure, i.e., when the trie enlarges its space such that $z \leq m \leq 2z - 1$ (where m is the number of elements it can maintain).

3.8.2 Practical Results

Having explained our LZ trie representations, we present a thorough experimental study of all tries for the LZ78 and LZW compression. The experiments are conducted with the tudocomp framework [69], whose strength is to facilitate (a) the implementation of compression algorithms and (b) the comparison between compression algorithms.

tudocomp. The framework is a lossless compression framework written in C++14. It supports building a pipeline of modules that transforms an input to a compressed binary output. This pipeline is flexible: appending, exchanging and removing a module in the pipeline in a plug-and-play manner is in the main focus of the design of tudocomp. A module can be further refined into submodules.

On the topmost abstraction level, tudocomp defines the abstract types `Compressor` and `Coder`. A *compressor* transforms an input into an output

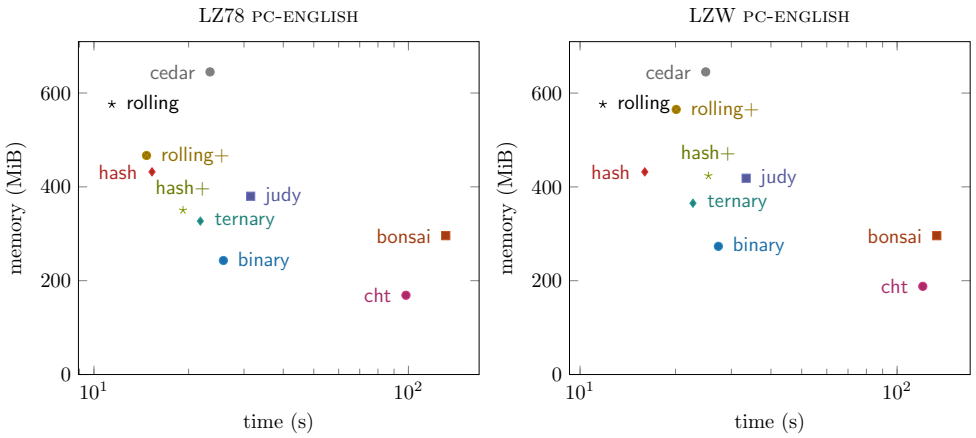


Fig. 3.42: Evaluation of LZ78 (*left*) and LZW (*right*) on PC-ENGLISH.

so that the input can be losslessly restored from the output by the corresponding *decompressor*. A *coder* takes an elementary data type (e.g., a character) and writes it to a compressed bit sequence. As with compressors, each coder is accompanied by a *decoder* taking care of restoring the original data from its compressed bit sequence. By design, a coder can take the role of a compressor, but a compressor may not be suitable as a coder (e.g., a compressor that needs random access on the whole input). Figure 3.41 shows a selection of compressors and coders available in the framework.

The behavior of a compressor or coder can be modified by passing different parameters. A parameter can be an elementary data type like an integer, but it can also be an instance of a class that specifies certain subtasks like integer coding. For instance, the compressors `lz78(trie, coder)` and `lzw(trie, coder)` take a dynamic trie data structure `trie` and a `coder` (to code an LZ78/LZW factor) as parameters. The first parameter `trie` can also be parametrized. For instance, hash requests a hash function and the load factor α. The coder is supplied as a parameter such that the respective compressor can call the coder directly (instead of alternatively piping the output of `lz78` or `lzw` to a coder). For our experiments, we selected the coder `bit` such that a call is `lz78(trie, bit)` or `lz78(trie, bit)`, where `trie` is one of the implemented trie data structures, e.g., binary.

Experimental setup. We implemented our LZ tries of Sect. 3.8.1 in the tudocomp framework.[20] As already mentioned before, we selected the coder `bit`. This coder stores the referred index y (with $y > 0$) of a factor F_x in $\lceil \lg x \rceil$ bits. That is because the factor F_x can have a referred index y only with $y < x$. We can restore the coded referred index on decompression since we know the index of the factor that we currently process and hence the number of bits used to

[20] The source code of our implementation is freely available at https://github.com/tudocomp, except for cht and bonsai due to copyright restrictions.

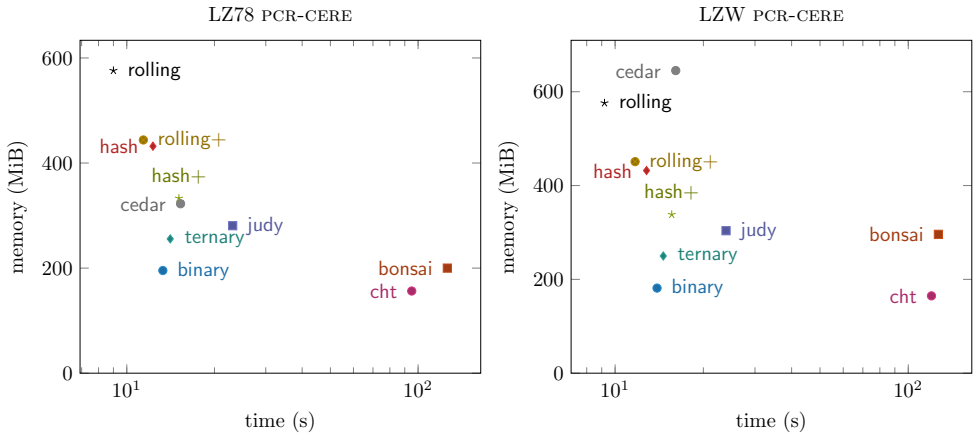


Fig. 3.43: Evaluation of LZ78 (*left*) and LZW (*right*) on PCR-CERE.

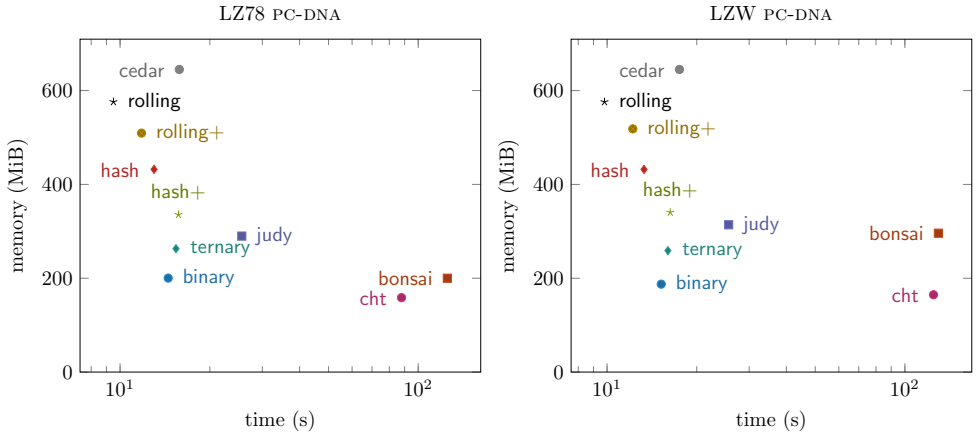


Fig. 3.44: Evaluation of LZ78 (*left*) and LZW (*right*) on PC-DNA.

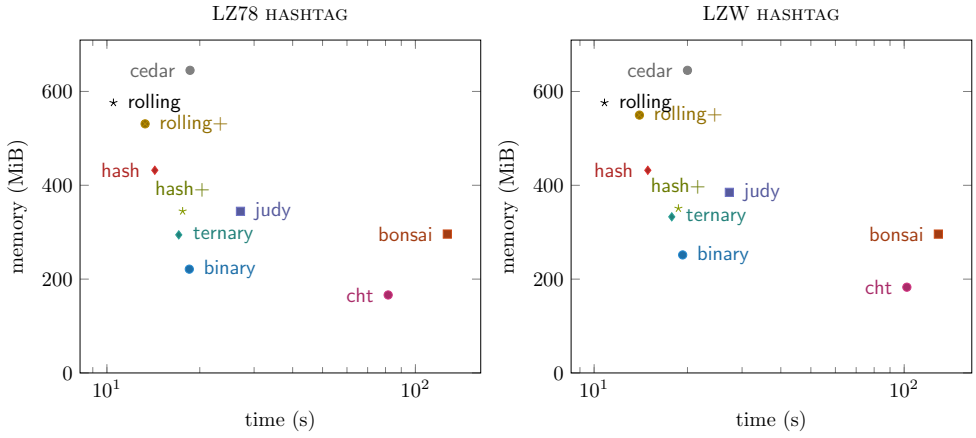


Fig. 3.45: Evaluation of LZ78 (*left*) and LZW (*right*) on HASHTAG.

store its referred index.[21] This yields $\sum_{i=1}^{z} \lceil \lg i \rceil = z \lceil \lg z \rceil - (\lg e)z + \mathcal{O}(\lg z)$ bits according to Stirling's formula for storing the (positive) referred indices.

For LZW, we have to cope with the negative integer values: We add the value σ to all output values such that its output consists of non-negative integers. Now the x-th factor costs $\lceil \lg(x + \sigma) \rceil$ bits. By splitting up the sum $\sum_{i=1}^{z} \lceil \lg(i + \sigma) \rceil = \sum_{i=1}^{z+\sigma} \lceil \lg i \rceil - \sum_{i=1}^{\sigma} \lceil \lg i \rceil$ we get the total number of bits of the LZW output by the previous formula. For LZ78, the additional characters are output naïvely as $\lceil \lg \sigma \rceil$-bit integers.

The LZ78 and LZW compressor are independent of the LZ trie implementation, i.e., all trie data structures described in the previous sections can be plugged into the LZW or LZ78 compressor easily. We additionally incorporated the following trie data structures into tudocomp:

cedar: the Cedar trie [245], representing a trie using two arrays.

judy: the Judy array, advertised to be optimized for avoiding cache misses (cf. [179] for an evaluation).

bonsai: the m-Bonsai (γ) trie [206] representing a trie whose nodes are not labeled. It uses a compact hash table, but unlike our approach, the key consists of the position of the parent in the hash table (instead of the label of the parent) and the character. Due to this fact, we need to traverse the complete trie for enlarging the trie. We store the labels of the trie nodes in an extra array.

All data structures are implemented as C++ classes. We added a lightweight wrapper around each class providing the same interface for all tries.

Filter. Given an input stream with known length, we evaluate the online computation of the LZ78 and LZW compression for different LZ trie representations. On computing a factor, we encode it and output it instantaneously. This makes our compression program a *filter* [188], i.e., it processes the input stream and generates an output stream, buffering neither the input nor the output.

Implementation details. The keys stored by hash are 40-bit integers, the fingerprints of rolling are 64-bit integers, and the values stored by hash, rolling and bonsai are 32-bit integers. For all variants working with hash tables, we initially set α to 0.3.

Hash function. We use cht with a hash function of the LCG family. Our hash table for hash uses an xorshift hash function[22] derived from [223]. It is

[21] This approach is similar to the description of an LZW file compressor found at http://www.cplusplus.com/articles/iL18T05o.

[22] http://xorshift.di.unimi.it/splitmix64.c

slower than simple multiplicative functions, but more resilient against clustering. Alternatives are sophisticated hash functions like CLHash [173] or Zobrist hashing [171, 248]. These are even more resilient against clustering, but have practical higher computation times in our experiments.

Datasets. We evaluated the combinations of the aforementioned tries with the LZW and LZ78 algorithms on the 200 MiB text collections described in Sect. 3.5.6. We assume that the input alphabet can be represented in bytes (i.e., $\sigma = 2^8$). The indices of the factors are represented with 32-bit integers. Figure 3.48 shows the number of factors produced by LZ78, LZW and LZ77 (we used the variant with overlapping) on each text collection. We plotted the memory consumption against the time (in logarithmic scale) for all datasets in Figs. 3.42 to 3.45. To avoid clutter, we selected one hash function per rolling hash table: We chose fermat with rolling and ID37 with rolling+ for the plots.

Overall evaluation. The evaluation shows that the fastest option is rolling. The size of its fingerprints is a trade-off between space and the probability of a correct output. When space is an issue, rolling with 64-bit fingerprints is not competitive with the other trie data structures. hash is the second fastest LZ trie in the experiments. With 40-bit keys it uses less memory than rolling, but is slightly slower. Depending on the quality of the resize hint, the variants hash+ and rolling+ take from 50% up to 100% of the size of hash and rolling, respectively. hash+ and rolling+ are always slower than their respective standard variants, sometimes slower than the deterministic data structures ternary and binary. binary's speed excels at texts with very small alphabets, while ternary outperforms binary texts with larger alphabets. Only cht can compete with binary in terms of space, but is slower by magnitudes than most alternatives. The third party data structures cedar, bonsai and judy could not make it to the Pareto front.

Evaluation of rolling. The hash table with the rolling hash function fermat is slightly faster than with a function of the ID37 family, but the hash table with fermat tends to have more collisions (cf. Fig. 3.46). It is significantly slower at less compressible texts like PC-PROTEINS due to the large number of collisions. The number of collisions can drop if we post-process the output of fermat with a hash function that is more collision resistant. Applying an evenly distributing hash function on fermat speeds up the computation only if the number of collisions is sufficiently high (e.g., rolling+ with fermat in Fig. 3.46). In the experiments, we apply the xorshift hash function used by hash to the output of fermat for determining the initial address. We denote this variant with a $\oplus$ as suffix of either fermat$\oplus$ or rolling$\oplus$.

According to the birthday paradox, the likelihood that the fingerprints of two different substrings match is anti-proportional to the number of bits used for

Trie	#Collisions	M	Memory	Time
rolling with				
- ID37	36 M	33.6 M	576.0 MiB	11.6 s
- fermat	137 M	33.6 M	576.0 MiB	11.4 s
- fermat⊕	36 M	33.6 M	576.0 MiB	11.8 s
rolling+ with				
- ID37	140 M	24.0 M	466.9 MiB	14.7 s
- fermat	938 M	24.0 M	466.9 MiB	21.0 s
- fermat⊕	142 M	24.0 M	466.9 MiB	15.8 s
hash	36 M	33.6 M	432.0 MiB	15.3 s
hash+	137 M	24.0 M	350.2 MiB	19.1 s

Fig. 3.46: Detailed evaluation of the tries using hashing (defined in Sect. 3.8.1). We evaluated the number of collisions and the final table size M (measured in millions) for the LZ78 factorization of 200 MiB PC-ENGLISH. An entry in rolling costs $64 + 32$ bits, an entry in hash $40 + 32$ bits.

storing a fingerprint if we assume that the used rolling hash function distributes uniformly. This means that the domain of the Karp-Rabin fingerprints can be made large enough to be robust against collisions when hashing large texts. In our case, we used 64-bit fingerprints because, unlike 32-bit and 40-bit fingerprints, the factorization produced by rolling are correct for all test instances and all tested rolling hash functions. Nevertheless, this bit length can be considered as too weak for processing massive datasets: Given that the used rolling hash function is uniform, the probability of a collision is $1/2^{64}$. Although this number is very small, processing 10^9 datasets, each 200 MiB large, would give a collision probability of roughly 1%. This probability can be reduced by enlarging the bit length, and hence improving the correctness probability by sacrificing working space. We reran our experiments with 64-bit and 128-bit fingerprints, and measured time and space usage in Fig. 3.49. There, we can see that switching to a larger bit length slightly degrades the running time, but severely degrades the space usage.

Another option to sustain a correct computation is to check the output factorization. This check can be done by reconstructing the text with the built LZ trie. However, a compression with rolling combined with a decompression step takes more time than other approaches like hash or binary. Hence, a Las Vegas algorithm based on rolling is practically not interesting.

Variations of Hash Tables. The trie representation hash can be generalized to be used with any associative container. The easiest implementation is to use the balanced binary tree `std::map` or the hash table `std::unordered_map` provided by the standard library of C++11. The hash table `std::unordered_map` is conform to the interface of the C++ standard library, but therefore sacrifices performance. It uses separate chaining that tends to use a lot of small memory allocations affecting the overall running time (see Fig. 3.50). Another pitfall is to use the standard C++11 hash function for integers that is just the identity

function. Although this is the fastest available hash function, it performs poorly in the experiments. There are two reasons. The first is that $k \mapsto k \mod M$ badly distributes the tuples if M is not a prime. The second is that the input data is not independent: In the case of LZ78 and LZW, the composed key $c + \ell\sigma$ of a node v connected to its parent with label ℓ by an edge with label c holds information about the trie topology: all nodes whose keys are $\ell\sigma + d$ for a $d \in \Sigma$ are the siblings of v. Since ℓ is smaller than the label of v (ℓ is the referred index of the factor corresponding to v), larger keys depend on the existence of some keys with smaller values. Both problems can be tackled by using a hash function with an avalanche effect property, i.e., flipping a single bit of the input changes roughly half of the bits of the output. In Fig. 3.50 we evaluated the identity and the xorshift hash function as hash functions for the hash table `flathash`, which seems to be very sensitive for hash collisions. We selected the LZ trie of the LZ-index [193] as an external competitor in Fig. 3.50: We terminated the execution of the LZ-index algorithm after producing the LZ trie of the LZ78 factorization. We did not integrate this data structure into tudocomp. The only interesting configuration is hash with the hash table `sparsehash`, since it takes 4.1 MiB less space than binary while still being faster than cht at the LZ78 factorization of PCR-CERE.

3.9 Conclusion

We presented novel approaches for the LZ77 and the LZ78 factorizations. Our main idea was based on two different representations of the suffix tree that are especially trimmed on a small memory footprint during their construction. The text was only needed during the computation of our two suffix tree representations. For computing the factorizations we used head to retrieve a character of the text.

We found the LZ77 factorization to be valuable for various applications like computing the LPF table or all distinct squares. Although we were careful about the choice of the suffix tree representations, they are still the bottlenecks in terms of the working space. We therefore hope that our algorithms could work with less space in the light of future achievements in suffix tree construction algorithms. While tackling all the problems addressed in this chapter, we gathered plenty of open problems:

Rightmost parsing. The referred position of a referencing factor F is not uniquely determined by Def. 3.1. In terms of the suffix tree topology, we can choose the suffix number of all already visited leaves belonging to the subtree of F's witness. From all possible referred positions, our algorithms in Sect. 3.4 choose the smallest one. Many compression programs encode F's referred position j by the difference between F's starting position and j with a coder favoring small numbers (cf. [79]). Since a universal coder favors small

numbers, we can optimize its output by choosing the largest of all possible referred positions, also called the *rightmost* one. The best known algorithm is by Belazzougui and Puglisi [27]. Their algorithm runs in $\mathcal{O}(n(1+\lg\sigma/\sqrt{\lg n}))$ time while using $\mathcal{O}(n\lg\sigma)$ bits of working space. Unfortunately, it seems impossible to adapt our LZ77 factorization algorithms in Sect. 3.4 to compute the rightmost parsing without worsening the running time. It is clear that the rightmost parsing and the standard LZ77 factorization produce the same factors, since only the referred positions differ. This means that we only have to reason about how to find the referred positions.

An idea would be to use a semi-dynamic RMQ data structure on an array A, where A is allowed to have not-initialized entries, which can be initialized later. The idea is to create the array A of length n, and to built on top of A a semi-dynamic range *maximum* query (RMQ) data structure that retrieves the *maximum* entry within a given range. All entries of A are uninitialized at the beginning (the same as setting $A[i] \leftarrow -\infty$ for $1 \leq i \leq n$). In one single LZ77 pass, we can identify the referred positions while initializing entries of A with the suffix numbers of the visited leaves. Within this pass, we set $A[\textsf{leaf\_rank}(\lambda)] \leftarrow j$ after visiting a leaf λ with suffix number j. On visiting a leaf corresponding to a witness w, we query the semi-dynamic RMQ data structure for the entry with the maximum value in the range $[\textsf{leaf\_rank}(\textsf{lmost\_leaf}(w))\,..\,\textsf{leaf\_rank}(\textsf{rmost\_leaf}(w))]$. Unfortunately, we are not aware of any semi-dynamic RMQ data structure, tailored to the setting of (a) performing n updates, and (b) answering z queries efficiently. The closest solution is a fully dynamic RMQ data structure, for which Brodal et al. [39] presented a solution performing each operation within $\mathcal{O}(\lg n/\lg\lg n)$ amortized time.

An alternative approach would be to compute the rightmost parsing online with a modification of the algorithm of Gusfield [120, Sect. 7.17.1] that computes the non-overlapping LZ77 factorization. The idea of this algorithm is to build the suffix tree with Ukkonen's algorithm [236] online. Remembering how all distinct squares are computed online in Thm. 3.28, whenever this suffix tree construction algorithm creates a new leaf λ, the invariant holds that $\textsf{sufnum}(\lambda)$ is fixed (i.e., it will not change), and that all leaves with smaller suffix numbers are already present in the suffix tree. To compute the rightmost parsing, we only consider leaves corresponding to a referencing factor. Given such a leaf λ, our problem is to find the leaf λ' with the second largest suffix number among all leaves that are in the subtree of λ's parent u. That is because λ' had the *largest* suffix number among all leaves in u's subtree before adding λ. Approaches like a semi-dynamic RMQ data structure (supporting to set values only once) require an integer array, in which we can store the suffix numbers of the leaves. Unfortunately, this is only possible if we know the order of the suffixes in advance (and hence, the algorithm cannot work online any longer). It would be interesting to study ways on how to devise a semi-dynamic RMQ data structure on the suffix numbers of the suffix tree leaves that are created online.

LZ77 with $\mathcal{O}(z)$ words. Computing the LZ77 or LZ78 factorization within $\mathcal{O}(z)$ words of space in optimal time is still an open problem.[23] For nearly incompressible texts with $z = \Theta(n/\log_\sigma n)$, Cor. 3.14 already provides an algorithm running in $\mathcal{O}(n)$ time using $\mathcal{O}(z)$ words of space. For general z, there are two algorithms in literature that can work efficiently within $\mathcal{O}(z)$ words of working space:

The first algorithm is due to Fischer et al. [88], which approximates the LZ77 factorization: It computes at most $(1+\epsilon)z$ factors while using $\mathcal{O}(z)$ words of working space and $\mathcal{O}((n/\epsilon)\lg^2 n)$ time. We obtain the LZ77 factorization with two runs of the algorithm: In the first run, we set $\epsilon \leftarrow 1$ to obtain z' phrases with $z \leq z' \leq 2z$. For the second run, we set $\epsilon \leftarrow 1/(\delta z')$ for a real constant δ with $\delta > 1$. The second run produces $(1+\epsilon)z = z + z/(\delta z') < z + 1$ factors, i.e., it produces exactly the same number of factors as LZ77. The overall running time is $\mathcal{O}(nz\lg^2 n)$.

The second algorithm is due to Kosolobov [168]. It runs in $\mathcal{O}\big((n/\epsilon)\big(\lg\sigma + \lg((\lg n)/\epsilon)\big)\big)$ time with $\mathcal{O}(\epsilon n/\lg n)$ words of working space for a trade-off parameter ϵ with $0 < \epsilon < 1$. We determine an estimate of z with the first run of [88] as above and set $\epsilon \leftarrow (z\lg n)/n$. This gives $\mathcal{O}(n\lg^2 n + n^2(\lg\sigma + \lg(n/z))/(z\lg n))$ time and $\mathcal{O}(z)$ words of working space. This running time is better than $\mathcal{O}(nz\lg^2 n)$ time when $z = \Omega(\sqrt{n(\lg(n/z) + \lg\sigma)}/\lg^{3/2} n)$. In this case, the algorithm runs in $\mathcal{O}(n^{3/2}\lg n)$ time.

Engineering a practical LZ77 algorithm. The LZ77 algorithm does not use many properties of the suffix tree. One could build just SA, LCP, and a previous-/next-smaller-value data structure on top of LCP to support the parent-operation. In this suffix tree representation, nodes are represented as pairs of integers, known as LCP intervals [2]. The main challenge is to map these pairs to plain numbers such that we can mark, rank, and select nodes efficiently.

Computing the LPF table in linear time with compressed space. We wonder whether it is possible to compute the LPF table in linear time with $\mathcal{O}(n\lg\sigma)$ bits of space. The solution is easy when we are allowed to maintain an array X with $n\lg n$ bits, in which we will store the LPF table as a plain array at the end: In that case, we first construct SA in X like in the succinct suffix tree representation. Following Lemma 3.22, we can construct LPF in the representation of Cor. 3.21 in linear time. Finally, we copy LPF into X.

If we want to compute LPF within a working space of $\mathcal{O}(n\lg\sigma)$ bits, it seems hard to achieve linear running time. That is because we need access to the string depth of the suffix tree node w for each entry LPF[i], where w is the lowest node having the leaf λ with suffix number i and a leaf with a suffix number less than i in its subtree. Recall that we compute str_depth(w) by (a)

[23] The idea of this problem was worked out together with Moshe Lewenstein.

taking λ and another leaf λ' with $\mathsf{lca}(\lambda, \lambda') = w$ and (b) recursively applying $\mathsf{next\_leaf}$ on λ and λ' to finally move to leaves whose string labels differ in the first character. Let $v \coloneqq \mathsf{lca}(\mathsf{next\_leaf}(\lambda), \mathsf{next\_leaf}(\lambda'))$ be the node whose string label is yielded by removing the first character from the string label of w. Let w' be the lowest node having the leaf λ' with suffix number $i+1$ and a leaf with a suffix number less than $i+1$ in its subtree. If $w' = v$, then $\mathsf{LPF}[i+1] = \mathsf{LPF}[i]-1$. Otherwise, $\mathsf{str\_depth}(v) < \mathsf{str\_depth}(w')$. Suppose that we have computed $\mathsf{LPF}[i] = \mathsf{str\_depth}(w) = \mathsf{str\_depth}(v)+1$ (which can be computed in $\mathcal{O}(\mathsf{str\_depth}(w))$ time according to Fig. 3.9). Given that $\mathsf{LPF}[i+1] \geq \mathsf{LPF}[i]$, our goal is to compute $\mathsf{LPF}[i+1] = \mathsf{str\_depth}(w')$ in $\mathcal{O}(\mathsf{str\_depth}(v)-\mathsf{str\_depth}(w'))$ time by knowing $\mathsf{LPF}[i]$ and v (which is found by the algorithm described in Lemma 3.22). If we can achieve that, then it follows by Lemma 3.20 that we need $\mathcal{O}(n)$ time in total.

Memory efficient reversed LZ77 factorization. The space bounds of Thm. 3.37 computing the reversed LZ77 factorization (overlapping and non-overlapping) are due to a marked ancestor data structure. Actually, we deal with a specialized version of the marked ancestor problem: the semi-dynamic fringe marked ancestor problem[24], in which updates are restricted to marking a node that is a child of an already marked node. We seek for a succinct data structure that solves the semi-dynamic fringe marked ancestor problem by answering queries in amortized constant time, but spends $\mathcal{O}(\lg n / \lg \sigma)$ time for marking. Without the need of the $\mathcal{O}(n)$ words for the marked ancestor data structure, it becomes interesting to think how the parsing can be done with our two introduced suffix tree representations (recall Sect. 3.3.2), for which we have to explain how to conduct a pass in reverse text order. To reverse the order, we need the function $\mathsf{prev\_leaf}(\lambda)$ that returns (a) the leaf with suffix number $\mathsf{sufnum}(\lambda) - 1$ or (b) the leaf with the largest suffix number if $\lambda = \mathsf{smallest\_leaf}$. It can be computed as follows:

- With the succinct suffix tree, $\mathsf{prev\_leaf}(\lambda)$ is $\mathsf{leaf\_select}(\mathsf{ISA}[\mathsf{sufnum}(\lambda) - 1])$ if $\mathsf{sufnum}(\lambda) \geq 2$.
- With the compressed suffix tree, we can compute $\mathsf{prev\_leaf}(\lambda)$ with a backward search step with BWT introduced in the FM-index [78]. For the backward search we need, additionally to BWT, an array Acc that stores in $\mathsf{Acc}[\mathsf{c}]$ the first position i of SA with $T[\mathsf{SA}[i]] = \mathsf{c}$, for each character $\mathsf{c} \in \Sigma$. Fortunately, BWT and Acc are part of the compressed suffix tree [190], such that we can compute $\mathsf{prev\_leaf}(\lambda)$ with $\mathsf{Acc}[\mathsf{c}]+\mathsf{BWT.rank_c}(\mathsf{leaf\_rank}(\lambda)-1)$, where $\mathsf{c} \coloneqq \mathsf{BWT}[\mathsf{leaf\_rank}(\lambda)]$. The running time is $\mathcal{O}(t_{\text{rank}})$, where t_{rank} is the time for querying a wavelet tree (e.g., $t_{\text{rank}} = \mathcal{O}(\lg \lg \sigma)$ with the wavelet tree of Barbay et al. [23] using $\mathcal{O}(n \lg \sigma)$ bits of space).

[24] Coined by Breslauer and Italiano [36], who introduced the fully-dynamic variant of this problem.

Computing all distinct squares faster. There is one practical bottleneck for the offline variant, and one time-theoretical bottleneck of the online variant, for computing all distinct squares faster:

RMQs Remembering Sect. 3.5.5, our algorithm for computing all distinct squares depends on an RMQ data structure. Unfortunately, RMQ data structures are practically slow [202]. We wonder whether we can avoid the use of any RMQ without loosing linear running time.

Predecessor Dictionaries Theorem 3.28 describes the online algorithm running in $\mathcal{O}(n \lg^2 \lg n / \lg \lg \lg n)$ time, where the factor $\mathcal{O}(\lg^2 \lg n / \lg \lg \lg n)$ is due to the data structure of Beame and Fich [26]. This is also the current bottleneck of the online algorithm with respect to the running time. Future predecessor data structures could improve the overall performance of our online algorithm. For Weiner's suffix tree construction algorithm [241], it is possible to use the data structure of Fischer and Gawrychowski [84, Appendix A] to speed up a navigational operation to $\mathcal{O}(\lg^2 \lg \sigma / \lg \lg \lg \sigma)$ time. However, it is unclear whether the data structure can be used in conjunction with Ukkonen's algorithm, too.

Building the MAST in linear time. In Thm. 3.33, we left it open to compute the number of the non-overlapping occurrences of the string labels of the MAST nodes in linear time. A linear time algorithm computing these values would improve the running time of the greedy compression algorithm of Apostolico and Lonardi [10].

More sophisticated hashing techniques. Considering the trie representation with the hash table hash of Sect. 3.8.1.2, an interesting option is to switch from the linear probing scheme to a more sophisticated scheme whose running time is stable for high loads, too [180]. This could be especially beneficent if the resize hint provides more accurate bounds on the number of factors.

Speaking of novel hash tables, we can combine the compact hash table cht of Sect. 3.8.1.3 with the memory management of Google's sparse hash table[25], which leads us to the *compact sparse hash table*[26]. The compact sparse hash table is a blend of the techniques of both hash tables. It is more memory friendly than the compact hash table in case that the table is sparsely filled. The compact sparse hash table differs from cht in that we do not allocate an array with M entries. Instead, it consists of arrays that grow by doubling their sizes. The entries of the hash table are mapped to entries of the arrays with bit vectors.

In more detail, we partition the hash table in M/b sections, where b is a (small) constant. We assure that M is divisible by b such that all sections

[25] https://github.com/sparsehash/sparsehash
[26] https://github.com/tudocomp/compact_sparse_hash

have the same length b. For instance, this is the case when M and b are a power of two (with $b < M$). We store in an array of length M/b pointers to the sections. Given that we want to access the k-th element of the hash table ($1 \leq k \leq M$), there are integers i and j with $1 \leq i \leq b$ and $1 \leq j \leq M/b$ such that $k = i + (j-1)b$ (set $i := (k+b-1 \mod b) + 1$ and $j := \lceil k/b \rceil$). Then the k-th entry of the hash table corresponds to the i-th entry of the j-th section. The j-th section is represented by a bit vector B_j of length b and a dynamic array A_j. We maintain B_j and A_j such that the i-th entry of the j-th section is stored at position $B_\mathrm{j}.\mathsf{rank}_1(i)$ in A_j. For a sufficiently small b, the rank query can be answered with a single CPU instruction (of a modern CPU architecture) on the bit vector B_j without the need of a (dynamic) rank-support.

Suppose we want to insert an entry in the j-th section. We can do that by altering B_j and rearranging the elements in A_j. Rearranging A_j can be done efficiently if the elements of A_j (which can be at most b many) fit into the CPU cache. Whenever we want to insert an element into a full array, we double its size. Initially, all arrays A_j are empty. We need M bits and $(M/b)(\lg n + \lg b)$ bits for all bit vectors B_j and all arrays A_j (consisting of a pointer with $\lg n$ bits and a counter with $\lg b$ bits maintaining its size), respectively. We additionally need $(M/b) \lg n$ bits for the array of pointers to the sections, summing up to $(M/b)(2 \lg n + \lg b) + 3M$ bits of space in total, where the additional $2M$ bits are for the two bit vectors of the compact hash table by Cleary [50].

Dynamic succinct trie library. Current research articles [15, 140, 206] focus on dynamic, yet succinct, trie representations. A more wide-ranging idea than Sect. 3.9 is to bundle the trie data structures presented in Sect. 3.8 within a programming library focussing on dynamic succinct trie data structures. A new challenge is to additionally think about the *delete* operation, which is not needed for building the LZ trie. Asides from that, the trie data structures can be used to evaluate different flavors of LZ78 like [15, 117], which work with dynamic tries.

3.10 Landscape Oriented Figures

Collection	σ	$\max_{\text{LCP}}$	avg_{LCP}	BWT runs	z	$\max_x \lvert F_x \rvert$	H_0	H_3
HASHTAG	179	54,075	84	63,014 k	13,721 k	54,056	4.59	2.46
PC-DBLP.XML	97	1084	44	29,585 k	7035 k	1060	5.26	1.43
PC-DNA	17	97,979	60	128,863 k	13,970 k	97,966	1.97	1.92
PC-ENGLISH	226	987,770	9390	72,032 k	13,971 k	987,766	4.52	2.42
PC-PROTEINS	26	45,704	278	108,459 k	20,875 k	45,703	4.20	4.07
PCR-CERE	6	175,655	3541	10,422 k	1447 k	175,643	2.19	1.80
PCR-EINSTEIN.EN	125	935,920	45,983	153 k	496 k	906,995	4.92	1.63
PCR-KERNEL	161	2,755,550	149,872	2718 k	775 k	2,755,550	5.38	2.05
PCR-PARA	6	72,544	2268	13,576 k	1927 k	70,680	2.12	1.87
PC-SOURCES	231	307,871	373	47,651 k	11,542 k	307,871	5.47	2.34
TAGME	206	1281	26	65,195 k	13,841 k	1279	4.90	2.60
WIKI-ALL-VITAL	205	8607	15	80,609 k	16,274 k	8607	4.56	2.45

Fig. 3.47: Datasets of size 200 MiB used during the evaluations. We write 1 k for 10^3. The alphabet size σ includes the delimiter \$. The expressions $\max_{\text{LCP}}$ and avg_{LCP} are the maximum and the average value of LCP. The number of LZ77 factors is z. The number of runs consisting of one character in the BWT is called *BWT runs*.

		LZ78			LZW			LZ77	
Collection	σ	z	$z\lceil\lg(z\sigma)\rceil$	|output|	z	$z\lceil\lg(z+\sigma)\rceil$	|output|	z	$2z\lg n$
PC-ENGLISH	226	21.4 M	83.8 MiB	80.2 MiB	23.5 M	70.1 MiB	66.1 MiB	14.0 M	93.3 MiB
PCR-CERE	6	15.8 M	50.0 MiB	58.2 MiB	17.1 M	50.9 MiB	46.9 MiB	1.4 M	9.7 MiB
PC-DNA	17	16.4 M	54.8 MiB	60.5 MiB	17.8 M	52.9 MiB	48.9 MiB	13.9 M	92.1 MiB
HASHTAG	179	18.9 M	73.4 MiB	70.6 MiB	21.1 M	62.9 MiB	58.9 MiB	13.7 M	90.4 MiB

Fig. 3.48: Properties of the text collections and their factorizations. Each column |output| shows the size of the respective (compressed) output. The sizes $z\lceil\lg(z\sigma)\rceil \leq z\lceil\lg z\rceil + z\lceil\lg\sigma\rceil$ bits, $z\lceil\lg(z+\sigma)\rceil$ bits and $2z\lg n$ bits are the output size of the LZ78, LZW and LZ77 factorization for the respective number of factors z when storing the output in arrays of fixed bit width.

	LZ78				LZW			
	64 bit		128 bit		64 bit		128 bit	
	Time	Space	Time	Space	Time	Space	Time	Space
PC-ENGLISH (see also Fig. 3.42)								
rolling	11.4 s	576.0 MiB	12.1 s	960.0 MiB	11.8 s	576.0 MiB	12.7 s	960.0 MiB
rolling⊕	11.9 s	576.0 MiB	13.7 s	960.0 MiB	12.3 s	576.0 MiB	14.1 s	960.0 MiB
rolling+	21.0 s	466.9 MiB	24.1 s	778.1 MiB	68.8 s	565.1 MiB	52.6 s	984.6 MiB
rolling+⊕	15.8 s	466.9 MiB	18.3 s	778.1 MiB	24.9 s	565.1 MiB	22.5 s	984.6 MiB
PCR-CERE (see also Fig. 3.43)								
rolling	9.0 s	576.0 MiB	9.5 s	960.0 MiB	9.2 s	576.0 MiB	9.8 s	960.0 MiB
rolling⊕	9.5 s	576.0 MiB	10.8 s	960.0 MiB	9.6 s	576.0 MiB	11.2 s	960.0 MiB
rolling+	11.1 s	443.9 MiB	12.6 s	739.7 MiB	11.4 s	450.9 MiB	13.8 s	751.6 MiB
rolling+⊕	11.2 s	443.9 MiB	13.9 s	739.7 MiB	11.7 s	450.9 MiB	15.0 s	751.6 MiB
PC-DNA (see also Fig. 3.44)								
rolling	9.4 s	576.0 MiB	10.0 s	960.0 MiB	9.5 s	576.0 MiB	10.2 s	960.0 MiB
rolling⊕	9.8 s	576.0 MiB	11.5 s	960.0 MiB	10.0 s	576.0 MiB	11.6 s	960.0 MiB
rolling+	11.6 s	509.3 MiB	13.5 s	745.3 MiB	12.0 s	518.4 MiB	14.5 s	756.7 MiB
rolling+⊕	11.7 s	509.3 MiB	14.7 s	745.3 MiB	12.2 s	518.4 MiB	15.6 s	756.7 MiB
HASHTAG (see also Fig. 3.45)								
rolling	13.4 s	576.0 MiB	15.6 s	960.0 MiB	19.8 s	576.0 MiB	25.3 s	960.0 MiB
rolling⊕	10.8 s	576.0 MiB	12.6 s	960.0 MiB	11.1 s	576.0 MiB	12.9 s	960.0 MiB
rolling+	15.4 s	530.9 MiB	18.0 s	766.3 MiB	21.8 s	549.7 MiB	25.6 s	779.6 MiB
rolling+⊕	14.1 s	530.9 MiB	17.2 s	766.3 MiB	15.8 s	549.7 MiB	18.9 s	779.6 MiB

Fig. 3.49: Performance comparison of 64-bit and 128-bit fingerprints generated by fermat.

	PC-ENGLISH				PCR-CERE			
	LZ78		LZW		LZ78		LZW	
Trie	Time	Space	Time	Space	Time	Space	Time	Space
hash with hash table								
`std::unordered_map`	51.0 s	856.6 MiB	54.0 s	937.9 MiB	42.3 s	703.2 MiB	44.1 s	760.8 MiB
`std::map`	161.2 s	980.2 MiB	167.2 s	1.1 GiB	98.8 s	722.5 MiB	104.6 s	781.6 MiB
`rigtorp`[a]	14.9 s	960.0 MiB	15.2 s	960.0 MiB	12.0 s	960.0 MiB	12.3 s	960.0 MiB
`flathash`[b]	33.5 s	24 GiB	24.5 s	24 GiB	18.5 s	6 GiB	19.2 s	6 GiB
`flathash`[c]	15.1 s	1.3 GiB	15.7 s	1.3 GiB	12.4 s	1.3 GiB	13.0 s	1.3 GiB
`densehash`[d]	23.0 s	576.0 MiB	24.4 s	576.0 MiB	29.4 s	576.0 MiB	30.8 s	576.0 MiB
`sparsehash`[d]	49.1 s	255.7 MiB	52.2 s	280.0 MiB	68.6 s	191.3 MiB	72.4 s	206.1 MiB
LZ-index [193]	24.6 s	1047 MiB			14.5 s	817.3 MiB		

Fig. 3.50: LZ78 and LZW factorizations with the trie data structure hash combined with different hash tables. These approaches are compared with the LZ78 factorization of the LZ-index.

[a] https://github.com/rigtorp/HashMap with $\alpha = 0.5$ hard coded.
[b] https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/, it uses the identity as a hash function and doubles its size when experiencing too much collisions.
[c] See Footnote b, but with the xorshift hash function.
[d] https://github.com/sparsehash/sparsehash

Chapter

4

Sparse Suffix Sorting

A record, if it is to be useful to science,
must be continuously extended,
it must be stored,
and above all it must be consulted.
— Vannevar Bush [41]

Sorting suffixes of a long text lexicographically is an important first step for many text processing algorithms like the LZ factorization algorithms described in Chapter 3. The complexity of the problem is quite well understood (see [209]), as for integer alphabets suffix sorting can be done in optimal linear time and in-place [114, 177]. In this chapter, we consider a variant of this problem: instead of computing the order of *all* suffixes, we are content with sorting certain specified suffixes. This problem, called *sparse suffix sorting problem*, is formally defined as follows: Given a text $T[1\,..\,n]$ of length n and a set $\mathcal{P} \subseteq [1\,..\,n]$ of m arbitrary positions in T, the sparse suffix sorting problem asks for the (lexicographic) order of the suffixes starting at the positions in $\mathcal{P}$. The answer is encoded by a permutation of $\mathcal{P}$, which is called the *sparse suffix array (SSA)* of T (with respect to $\mathcal{P}$) and denoted by $\mathsf{SSA}(T, \mathcal{P})$.

Applications are found in external memory LCP array construction algorithms [141] and in the search of maximal exact matches [153, 238], i.e., substrings found in two given strings that can be extended neither to their left nor to their right without getting a mismatch.

Like the "full" suffix arrays, we can enhance $\mathsf{SSA}(T, \mathcal{P})$ with the lengths of the LCPs between adjacent suffixes in $\mathsf{SSA}(T, \mathcal{P})$. These lengths are stored in the *sparse longest common prefix array (SLCP)*, which we denote by $\mathsf{SLCP}(T, \mathcal{P})$. In combination, $\mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP}(T, \mathcal{P})$ store the same information as the *sparse suffix tree*, i.e., they implicitly represent a compacted trie over all suffixes starting at the positions in $\mathcal{P}$. The sparse suffix tree is an efficient index for pattern matching [164].

Based on classic suffix array construction algorithms [145, 200], sparse suffix sorting is easily conducted in $\mathcal{O}(n)$ time if $\mathcal{O}(n)$ words of additional working space are available. For $m = o(n)$, however, the working space may be too large, compared to the final space requirement of $\mathsf{SSA}(T, \mathcal{P})$. Although some special choices of $\mathcal{P}$ admit space-optimal $\mathcal{O}(m)$-words construction algorithms (e.g., [144], see also the related work listed in [33]), the problem of sorting arbitrary suffixes in small space seems to be much harder. We are aware of

Time	Space	Restriction	Ref.
$\mathcal{O}(n \lg n)$	$\mathcal{O}(n)$		[183]
$\mathcal{O}(n)$	$\mathcal{O}(n)$		[155]
$\mathcal{O}(n \lg n)$	$n + \mathcal{O}(1)$		[95]
$\mathcal{O}(n)$	$n + \mathcal{O}(1)$		[114, 177]
$\mathcal{O}(\tau m + n\sqrt{\tau})$	$\mathcal{O}(m + n/\sqrt{\tau})$		[145]
$\mathcal{O}(n)$	$\mathcal{O}(m)$	$\mathcal{P}$ evenly spaced	[144]
$\mathcal{O}(n \lg^2 m)$	$\mathcal{O}(m)$	MC	[33]
$\mathcal{O}(n \lg^2 m + m^2 \lg m)$	$\mathcal{O}(m)$	LV	[33]
$\mathcal{O}(n)$	$\mathcal{O}(m)$	MC	[104]
$\mathcal{O}(n\sqrt{\lg m})$	$\mathcal{O}(m)$	LV	[104]
$\mathcal{O}(n \lg m)$	$\mathcal{O}(m)$	MC or LV	[130]
$\mathcal{O}(n)$	$\mathcal{O}(m \lg m)$	MC	[130]
$\mathcal{O}(n + m \lg^2 n)$	$\mathcal{O}(m)$	MC, restore model	[208]

Fig. 4.1: Sparse suffix sorting algorithms. MC and LV denote Monte Carlo and Las Vegas algorithms, respectively. The trade-off parameter τ is in the domain $[1, \sqrt{n}]$. The space is measured in words, where it is assumed that an integer with $\lg n$ bits fits in one word.

the following results: As a deterministic algorithm, Kärkkäinen et al. [145] gave a trade-off using $\mathcal{O}(\tau m + n\sqrt{\tau})$ time and $\mathcal{O}(m + n/\sqrt{\tau})$ words of working space, where τ is a trade-off parameter with $1 \leq \tau \leq \sqrt{n}$. If randomization is allowed, there is a technique based on Karp-Rabin fingerprints, first proposed by Bille et al. [33] and later improved by I et al. [130]. Gawrychowski and Kociumaka [104] presented an algorithm running with $\mathcal{O}(m)$ words of additional space in either $\mathcal{O}(n\sqrt{\lg m})$ expected time as a Las Vegas algorithm, or in $\mathcal{O}(n)$ expected time as a Monte Carlo algorithm. Most recently, Prezza [208] presented a Monte Carlo algorithm in the restore model [47] that runs with $\mathcal{O}(m)$ words of space in $\mathcal{O}(n + m \lg^2 n)$ expected time. Figure 4.1 summarizes the running times and the memory usage of the listed algorithms.

4.1 Algorithm Outline and Our Contribution

We devise our sparse suffix sorting algorithm in the *restore model* [47], where algorithms are allowed to overwrite parts of the input, as long as they can restore the input to its original form at termination. In the case of sparse suffix sorting, we assume that the text T is stored as a rewritable array of size $n \lg \sigma$ bits in RAM. Apart from this space, we are only allowed to use $\mathcal{O}(m)$ words. The positions in $\mathcal{P}$ are assumed to arrive on-line, implying in particular that they need not be sorted. We aim at worst-case efficient *deterministic* algorithms:

Our main algorithmic idea is to insert the suffixes starting at the positions

of $\mathcal{P}$ into a self-balancing binary search tree [134]; since each insertion invokes $\mathcal{O}(\lg m)$ suffix-to-suffix comparisons, the time complexity is $\mathcal{O}(t_S m \lg m)$, where t_S is the cost for a suffix-to-suffix comparison. If all suffix-to-suffix comparisons are conducted naïvely by comparing the characters ($t_S = \mathcal{O}(n/\log_\sigma n)$ in the word random-access memory or machine (RAM) model), the resulting worst case time complexity is $\mathcal{O}(nm \lg m/\log_\sigma n)$. In order to speed this up, our algorithm identifies large identical substrings at different positions during different suffix-to-suffix comparisons. Instead of performing naïve comparisons on identical parts over and over again, we build a data structure (stored in redundant text space) to accelerate subsequent suffix-to-suffix comparisons. Informally, when two (possibly overlapping) substrings in the text are detected to be the same, one of them can be overwritten.

To accelerate suffix-to-suffix comparisons, we devise a new data structure called *hierarchical stable parsing (HSP) tree* that is based on the *edit sensitive parsing (ESP)* [55]. HSP trees support LCE queries and are *mergeable*, allowing us to build a dynamically growing LCE index on substrings read in the process of the sparse suffix sorting. Consequently, comparing two already indexed substrings is done by a single LCE query.

In their plain form, HSP trees need more space than the text itself; to overcome this space problem, we devise a *truncated* version of the HSP tree, yielding a trade-off parameter between space consumption and LCE query time. By choosing this parameter appropriately, the truncated HSP tree fits into the text space. With a text space management specialized on the properties of the HSP, we achieve the result of Thm. 4.1 below.

We make the following definition that allows us to analyze the running time more accurately. Define $\mathcal{C} \coloneqq \bigcup_{p,p' \in \mathcal{P}, p \neq p'} [p \,..\, p + \mathsf{lcp}(T[p\,..], T[p'\,..])]$ as the set of positions that must be compared for distinguishing the suffixes starting at the positions of $\mathcal{P}$. Then sparse suffix sorting is trivially lower bounded by $\Omega(|\mathcal{C}| \,/\, \log_\sigma n)$ time. With the definition of $\mathcal{C}$, we now can state the main result of this chapter as follows:

Theorem 4.1. Given a text T of length n that is loaded into RAM, the SSA and SLCP of T for a set of m arbitrary positions can be computed deterministically in $\mathcal{O}(|\mathcal{C}|\,(\sqrt{\lg \sigma} + \lg\lg n) + m \lg m \lg n \lg^* n)$ time, using $\mathcal{O}(m)$ words of additional working space.

Excluding the loading cost for the text, the running time can be sublinear (when $|\mathcal{C}| = o(n/(\sqrt{\lg \sigma} + \lg\lg n))$ and $m \lg m = o(n/\lg n \lg^* n)$). To the best of our knowledge, this is the first algorithm that refines the worst-case performance guarantee. All previously mentioned (deterministic and randomized) algorithms take $\Omega(n)$ time even if we exclude the loading cost for the text. Also, general string sorters (e.g., forward radix sort [8] or multikey quicksort [30]), which do not take advantage of the overlapping of suffixes, suffer from the lower bound of $\Omega(\ell/\log_\sigma n)$ time, where ℓ is the sum of all LCP values in the SLCP, which is always at least $|\mathcal{C}|$, but can in fact be $\Theta(nm)$.

As a result of independent interest, we uncover a flaw in the approximation bound of the algorithm of Cormode and Muthukrishnan [55] computing the *string edit distance with moves (SEDM)* approximatively. There, the authors postulated that they can approximate the SEDM of two strings of length n with a factor of $\mathcal{O}(\lg n \lg^* n)$ with ESP trees. However, there is a flaw in their analysis of the ESP trees. This flaw leads us to the discovery that the approximation factor is $\Omega(\lg^2 n)$ in worst case.

4.1.1 Suffix Sorting and LCE Queries

The LCE problem is to preprocess a text T such that subsequent LCE queries can be answered efficiently. Data structures for LCE and sparse suffix sorting are closely related, as shown in the following observation:

Observation 4.2. Given a data structure that answers LCE queries in $\mathcal{O}(\tau)$ time for $\tau > 0$, we can compute sparse suffix sorting for m positions in $\mathcal{O}(\tau m \lg m)$ time by inserting suffixes into a balanced binary search tree. Conversely, given an algorithm computing the SSA and the SLCP of a text T of length n for m positions in $\mathcal{O}(f(n,m))$ time with $\mathcal{O}(m)$ words of space for a function f, we can construct a data structure in $\mathcal{O}(\max(f(n,m), n/m))$ time with $\mathcal{O}(m)$ words of space, answering LCE queries on T in $\mathcal{O}(n^2/m^2)$ time.

Proof. The first claim is due to Lemma 2.6. For the second claim, we use the data structure of [31, Thm. 1a] that answers LCE queries in $\mathcal{O}(\tau)$ time. The data structure uses the SSA and SLCP values of those suffixes whose starting positions are in a difference cover sampling modulo τ. This difference cover consists of $\mathcal{O}(n/\sqrt{\tau})$ text positions, and can be computed in $\mathcal{O}(\sqrt{\tau})$ time [52]. We obtain the claimed bounds on time and space by setting $\tau := n^2/m^2$. □

There has been a great interest in devising deterministic LCE data structures with trade-off parameters (see Fig. 4.2), or in compressed space [128, 198, 231]. One of the currently best data structures with a trade-off parameter is due to Tanimura et al. [230], using $\mathcal{O}(n/\tau)$ words of space and answering LCE queries in $\mathcal{O}(\tau \lg \min(\tau, n/\tau))$ time, for a trade-off parameter τ with $1 \leq \tau \leq n$. However, this data structure has a preprocessing time of $\mathcal{O}(n\tau)$, and is thus not helpful for sparse suffix sorting. We develop a new data structure for LCE with the following properties.

Theorem 4.3. There is a *deterministic* data structure using $\mathcal{O}(n/\tau)$ words of space that answers an LCE query $\ell := \mathsf{lce}(i,j)$ for two text positions i and j with $1 \leq i, j \leq n$ on a text of length n in $\mathcal{O}(\lg^* n(\lg(\ell/\tau) + \tau^{\lg 3}/\log_\sigma n))$ time, where $1 \leq \tau \leq n$. We can build the data structure in $\mathcal{O}(n(\lg^* n + (\lg n)/\tau + (\lg \tau)/\log_\sigma n))$ time with additional $\mathcal{O}(\max(n/\lg n, \tau^{\lg 3} \lg^* n))$ words during construction.

Construction		Data Structure		
Time	Working Space	Space	Query Time	Ref.
$\mathcal{O}(n\tau)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\tau \lg \min\left(\tau, \frac{n}{\tau}\right)\right)$	[230]
$\mathcal{O}(n^{2+\epsilon})$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(\tau)$	[32]
$\mathcal{O}\left(n\left(\lg^* n + \frac{\lg n}{\tau} + \frac{\lg \tau}{\log_\sigma n}\right)\right)$	$\mathcal{O}\left(\max\left(\frac{n}{\lg n}, \tau^{\lg 3} \lg^* n\right)\right)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\lg^* n \left(\lg\left(\frac{\ell}{\tau}\right) + \frac{\tau^{\lg 3}}{\log_\sigma n}\right)\right)$	Thm. 4.3
$\mathcal{O}\left(n\left(\lg^* n + \frac{\lg n}{\tau} + \frac{\lg \tau}{\log_\sigma n}\right)\right)$	$\mathcal{O}\left(\tau^{\lg 3} \lg^* n\right)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}\left(\lg^* n \left(\lg\left(\frac{n}{\tau}\right) + \frac{\tau^{\lg 3}}{\log_\sigma n}\right)\right)$	Cor. 4.33

Fig. 4.2: Deterministic LCE data structures with trade-off parameters, where ϵ with $\epsilon > 0$ is a constant, and τ with $1 \leq \tau \leq n$ is a trade-off parameter. The length returned by an LCE query is denoted by ℓ. Space is measured in *words*. The column *Working Space* lists the working space needed to construct a data structure, whereas the column *Space* lists the final space needed by a data structure.

The construction time of our data structure has an upper bound of $\mathcal{O}(n \lg n)$, and hence it can be constructed faster than the deterministic data structures in [230] when $\tau = \Omega(\lg n)$.

4.1.2 Outline of this Chapter

We start with Sect. 4.2 introducing the ESP, where we conduct a thorough analysis on its characteristics for comparing two substrings by their ESP trees. Within this analysis we encounter some drawbacks of the ESP in Sect. 4.2.4, among others the aforementioned flaw for approximating the SEDM problem. These drawbacks are our motivation for presenting our novel HSP, whose description follows in Sect. 4.3. There, it is demonstrated that HSP is immune to the flaw of the ESP. Subsequently, Sect. 4.3.3 shows the general techniques for answering LCE queries with the HSP tree. This is followed by Sect. 4.4 introducing our algorithm for the sparse suffix sorting problem with an abstract data type *dynamic LCE data structure (dynLCE)* that supports LCE queries and a merging operation. The remainder of that section shows that the HSP tree from Sect. 4.3 fulfills all properties of a dynLCE; in particular, HSP trees support the merging operation. The last part of this chapter is dedicated to the study on how the text space can be exploited with the HSP technique to improve the memory footprint. This leads us to truncated HSP trees with a merging operation that is tailored to working in text space (Sect. 4.5). With the truncated HSP trees we finally solve the sparse suffix sorting problem in the time and space as claimed in Thm. 4.1.

4.2 Edit Sensitive Parsing

The crucial technique used in this chapter is the *alphabet reduction*. The alphabet reduction is used to partition a string deterministically into blocks. The first work introducing the alphabet reduction technique to the string context was done by Mehlhorn et al. [189], who called their approach *signature encoding*. The signature encoding is derived from a tree coloring approach [112]. It supports string equality checks in the scenario where strings can be dynamically concatenated or split. In the same context, Sahinalp and Vishkin [220] studied the maximal number of characters to the left and to the right of a substring Z of Y such that changing one of these characters affects how Z is parsed by the signature encoding of Y. In a later work, Alstrup et al. [5] enhanced signature encoding with additional queries like LCE. Recently, an LCE data structure using signature encoding in compressed space was shown by Nishimoto et al. [198]. The most recent approach on signature encoding is by Gawrychowski et al. [109] presenting a mergeable LCE data structure. A slightly modified version of signature encoding is proposed by Sakamoto et al. [221]. They used

the alphabet reduction to build a grammar compressor that is approximating the size of the smallest grammar by a factor of $\mathcal{O}(\lg^* n \lg n)$.

A modified parsing was introduced by Cormode and Muthukrishnan [55]. They modified the parsing by restricting the block size from two up to three characters, and named their technique ESP. Initially used for approximating the SEDM, the ESP technique has been found to be applicable to building self-indexes [228]. We stick to the ESP technique, because the size of the subtree of a node in the ESP tree is bounded. In this section, we first introduce the ESP technique, and then give a motivation for a modification of the ESP technique, which we call HSP. Before that, we recall the alphabet reduction and the ESP trees.

4.2.1 Alphabet Reduction

Given a string Y in which no two adjacent characters are the same, i.e., $Y[i-1] \neq Y[i]$ for every integer i with $2 \leq i \leq |Y|$, we can partition Y (except at most the first $\lg^* \sigma$ positions) into *blocks* of size two or three with a technique called *alphabet reduction* [55, Sect. 2.1.1]. It consists of three steps (see also Fig. 4.3): First, it reduces the alphabet size to at most eight, in which every character has a rank from zero to seven. Subsequently, it substitutes characters with ranks four to seven with characters having a rank between zero and two. By doing so, it shrinks the alphabet size to three. Finally, it identifies certain text positions as landmarks that determine the block boundaries.

For reducing the alphabet size, we assume that $\sigma \geq 9$, otherwise we skip this step. The task is to generate a surrogate string Z on the alphabet $\{0, 1, 2\}$ such that the entry $Z[i]$ depends only on the substring $Y[i \mathinner{..} i + \lg^* \sigma]$, for $1 \leq i \leq |Y| - \lg^* \sigma$. To this end, we interpret Y as an array of binary strings, i.e., we interpret the character $Y[i]$ with its binary representation $Y[i] \in \{0, 1\}^*$. By doing so, we have $Y[i][\ell] \in \{0, 1\}$ for all integers ℓ with $1 \leq \ell \leq \lceil \lg \sigma \rceil$. We create an array Z of length $|Y|-1$ storing integers of the domain $[0 \mathinner{..} 2\lceil \lg \sigma \rceil - 1]$. For each text position i with $2 \leq i \leq |Y|$, we compare $Y[i]$ with $Y[i-1]$: We compute $\ell \coloneqq \mathsf{lcp}(Y[i-1], Y[i])$, and write $2\ell + Y[i][\ell + 1]$ to $Z[i]$ (remember that we treat $Y[i]$ as a binary string). By doing so, no two adjacent integers are the same in Z [55, Lemma 1]. Having computed Z, we recurse on Z until Z stores integers of the domain $\{0, \ldots, 5\}$. Note that the alphabet cannot be reduced further with this technique, since $2\lceil \lg x \rceil \geq x$ for every integer x with $2 \leq x \leq 6$. To obtain the final Z, we recurse at most $\lg^* \sigma$ times. Let r be the number of recursions. Then we have $|Y| = |Z| + r$.

If we skipped this step because of a small alphabet size ($\sigma \leq 8$), then we set $Z[i]$ to the rank of $Y[i]$ induced by the linear order of Σ (e.g., $Z[i] = 0$ if $Y[i]$ is the smallest character, similar to Cor. 2.9). Since $|Y| = |Z|$, we set r to zero.

To reduce the domain further, we iterate over the values $j = 3, \ldots, 8$ in ascending order, substituting each $Z[i] = j$ with the lowest value of $\{0, 1, 2\}$

that does not occur in its neighboring entries ($Z[i-1]$ and $Z[i+1]$, if they exist). Finally, Z contains only numbers between zero and two.

In the final step we create the *landmarks* that determine the block boundaries. The landmarks obey the property that the distance between two subsequent landmarks is greater than one, but at most three. They are determined by local maxima and minima: First, each number $Z[i]$ that is a local maximum is made into a landmark. Second, each local minimum that is not yet neighbored by a landmark is made into a landmark.

Finally, we create blocks by associating each position in Z with its closest landmark. Positions associated with the same landmark are put into the same block. As a tie breaking rule we favor the right landmark in case that there are two closest landmarks. The last thing to do is to map each block covering $Z[i..j]$ to $Y[i+r..j+r]$.

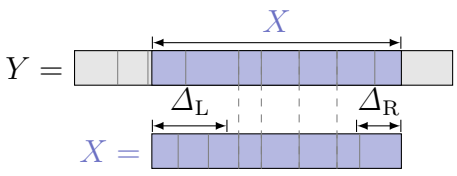
The tie breaking rule can cause a problem when $Z[1]$ and $Z[3]$ are landmarks, i.e., the leftmost block contains only one character. We circumvent this problem by fusing the blocks of the first and second landmark to a single block. If this block covers four characters, we split it evenly.

Altogether, the alphabet reduction needs $\mathcal{O}(|Y| \lg^* \sigma)$ time, since we perform $r \leq \lg^* \sigma$ reduction steps, while determining the landmarks and computing the blocks take $\mathcal{O}(|Y|)$ time. The steps are summarized in the following lemma:

Lemma 4.4. Given a string Y in which no two adjacent characters are the same, the alphabet reduction applied on Y partitions Y into blocks, except at most $\lceil \lg^* \sigma \rceil$ positions at the left. It runs in $\mathcal{O}(|Y| \lg^* \sigma)$ time.

The main motivation of introducing the alphabet reduction is the following lemma that shows that applying the alphabet reduction on a text Y and on a pattern X generates the same blocks in X as in all occurrences of X in Y, except at the left and right borders of a specific length:

Lemma 4.5 ([55, Lemma 4])**.** Given a substring X of a string Y in which no two adjacent characters are the same, the alphabet reduction applied to X alone creates the same blocks as the blocks representing the substring X in Y, except for at most $\Delta_L := \lceil \lg^* \sigma \rceil + 5$ characters at the left border, and $\Delta_R := 5$ characters at the right border.



Given a block β, we call the substring $Y[\mathsf{b}(\beta) - \Delta_L .. \mathsf{e}(\beta) + \Delta_R]$ the *local surrounding* of β, if it exists (i.e., $\mathsf{b}(\beta) - \Delta_L \geq 1$ and $\mathsf{e}(\beta) + \Delta_R \leq |Y|$). Blocks whose local surroundings exist are also called *surrounded*. A consequence of Lemma 4.5 is the following: Given that X is the local surrounding of a surrounded block β, then the blocking of every occurrence of X in Y is the

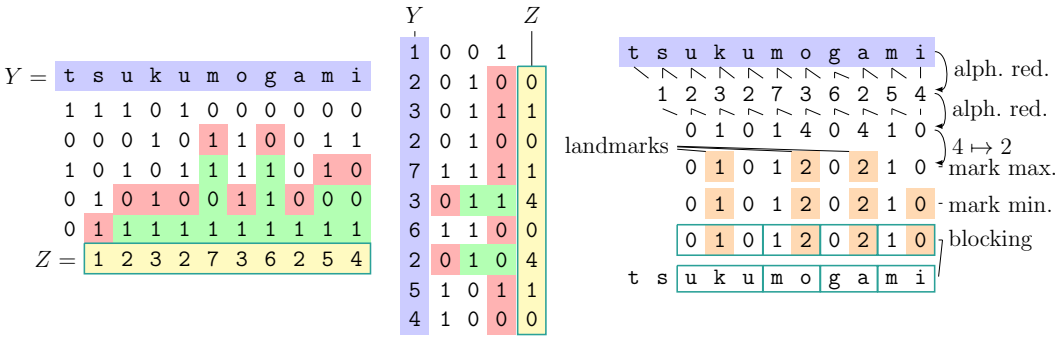


Fig. 4.3: Alphabet reduction applied on the string $Y =$ `tsukumogami`. We represent the characters with the five lowest bits of the ASCII encoding. *Left:* A single step of the alphabet reduction. The bit representation of each character $Y[i]$ is shown vertically in the left figure (the most significant bit is on the top). The alphabet reduction matches the least significant bits (shaded green) of two adjacent entries, and returns twice the number of matched bits plus the mismatched bit of the right character (shaded red). The resulting integer array Z is the last row. *Middle:* The second step of the alphabet reduction, where the result of the first alphabet reduction stored in Z is put into Y. *Right:* Computation of the blocks. Two steps of the alphabet reduction (seen in the left and in the middle image) yield a sequence consisting only of integers within the domain $\{0, \ldots, 4\}$. Subsequently, all '4's are replaced (in this case by '2' since the neighboring values are '0' and '1' in both cases), and the maxima and certain minima are made into landmarks (shaded orange). Finally, the boxes in the last two rows are the computed blocks.

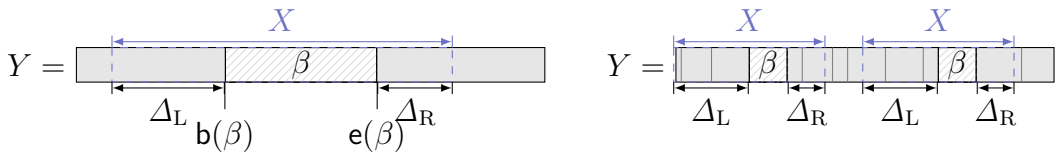


Fig. 4.4: *Left*: Surrounded block β with local surrounding X contained in a string Y. *Right*: Two occurrences of the local surrounding X of a surrounded block β in the string Y, which is partitioned into blocks (gray rectangles) by the edit sensitive parsing. Although the occurrences of X can be differently blocked at their borders, they all have a block equal to β in common.

same, except at most Δ_L and Δ_R characters at the left and right borders, respectively. We conclude that the blocking of every occurrence of X has a block $X[1 + \Delta_L \,..\, \Delta_L + |\beta|]$ that is equal to $Y[\mathsf{b}(\beta) \,..\, \mathsf{e}(\beta)]$ (see Fig. 4.4).

4.2.2 Meta-Blocks

Whenever a string Y contains a repetition of a character at two adjacent positions, we cannot parse Y with the alphabet reduction. A solution is to additionally use an auxiliary parsing specialized on repetitions of the same character. With this auxiliary parsing, we can partition Y into substrings, where each substring is either parsed with the alphabet reduction, or with the auxiliary parsing. It is this auxiliary parsing where the aforementioned signature encoding and the *edit sensitive parsing (ESP)* technique differ. The main difference is that the ESP technique restricts the lengths of the blocks: It first identifies so-called *meta-blocks* in Y, and then further refines these meta-blocks into blocks of length 2 or 3. The meta-blocks are created in the following 3-stage process (see also Fig. 4.5 for an example):

(1) Identify runs with smallest period one (i.e., maximal substrings of the form c^ℓ for $c \in \Sigma$ and $\ell \geq 2$). Such substrings form the Type 1 meta-blocks.

(2) Identify remaining substrings of length at least two (which must be bordered by Type 1 meta-blocks). Such substrings form the Type 2 meta-blocks.

(3) Every substring not yet covered by a meta-block consists of a single character and cannot have Type 2 meta-blocks as its neighbors. Such characters are fused with a neighboring meta-block. The meta-blocks emerging from this fusing are called Type M (mixed).

Meta-blocks of Type 1 and Type M are collectively called *repeating meta-blocks*. For (3), we are free to choose whether a remaining character should be fused with its preceding or succeeding meta-block (both meta-blocks are repeating). We stick to the following tie breaking rule:

Rule (M): Fuse a remaining character $Y[i]$ with its succeeding[1] meta-block, or, if $i = |Y|$, with its preceding meta-block.

Meta-blocks are further partitioned into *blocks*, each containing two or three characters from Σ. Blocks inherit the type of the meta-block they are contained in. How the blocks are partitioned depends on the type of the meta-block:

[1] The original version [55] prefers the preceding meta-block. We comply with Rule (M) as it behaves better. See Fig. 4.45 for an example with the later introduced HSP trees.

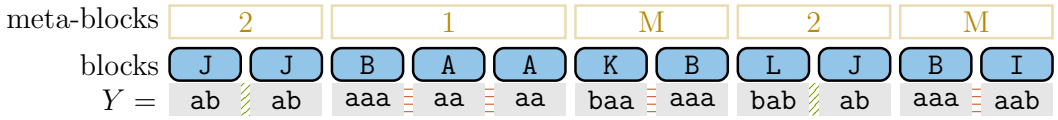


Fig. 4.5: ESP of the string $Y = \texttt{ababaaaaaaabaaaaababababaaaaab}$. The string is divided into blocks represented by the gray rectangular boxes at the bottom. Each block gets assigned a new character represented by the capital letters in the rounded boxes. The white/golden (□) rectangular boxes on the top level represent the meta-blocks that group the blocks. Each such box is labeled with the type of its respective meta-block. The blocks are connected with red horizontal lines (▤) if they belong to a repeating meta-block, or by green diagonal lines (▨) if they belong to a Type 2 meta-block.

Repeating meta-blocks. A repeating meta-block is partitioned greedily: create blocks of length three until there are at most four, but at least two characters left. If possible, create a single block of length two or three; otherwise (there are four characters remaining) create two blocks, each containing two characters.

Type 2 meta-blocks. A Type 2 meta-block μ is partitioned into blocks in $\mathcal{O}(|\mu| \lg^* \sigma)$ time by the alphabet reduction (Lemma 4.4). A block β generated by the alphabet reduction is determined by the characters $Y[\max(\mathsf{b}(\beta) - \Delta_{\mathrm{L}}, \mathsf{b}(\mu)) \mathinner{..} \min(\mathsf{e}(\beta) + \Delta_{\mathrm{R}}, \mathsf{e}(\mu))]$ due to Lemma 4.5. Given the number of reduction steps r in Sect. 4.2.1, the alphabet reduction does not create blocks for the first r characters of each meta-block. The ESP technique blocks the first r characters in the same way as a repeating meta-block. The border case $r = 1$ (one character remaining) is treated by fusing the remaining character with the first block created by the alphabet reduction, possibly splitting this block in the case that its size is four.

A block is called *repetitive* if it contains the same characters. All blocks of a Type 1 meta-block and all blocks except at most the left- or rightmost block (these blocks can contain a fused character) in a Type M meta-block are repetitive.

Let $\mathsf{esp}\colon \Sigma^* \to (\Sigma^2 \cup \Sigma^3)^*$ denote the function that parses a string by the ESP technique. We regard the output of esp as a string of blocks.

4.2.3 Edit Sensitive Parsing Trees

Applying esp recursively on its output generates a *context free grammar (CFG)* as follows. Let $\langle Y \rangle_0 := Y$ be a string on an alphabet $\Sigma_0 := \Sigma$. The output of $\langle Y \rangle_h := \mathsf{esp}^{(h)}(Y) = \mathsf{esp}(\mathsf{esp}^{(h-1)}(Y))$ is a sequence of blocks, which belong to a new alphabet Σ_h with $h \geq 1$. We call the elements of Σ_h with $h \geq 1$ *names*, and use the term *symbol* for an element that is a name or a character. A block $\beta \in \Sigma_h$

Common Dictionary	
Rule	string(·)
$\mathtt{I} \to \mathtt{aab}$	$\mathtt{a}^2\mathtt{b}$
$\mathtt{J} \to \mathtt{ab}$	$\mathtt{ab}$
$\mathtt{K} \to \mathtt{baa}$	$\mathtt{ba}^2$
$\mathtt{L} \to \mathtt{bab}$	$\mathtt{bab}$
$\mathtt{N} \to \mathtt{ba}$	$\mathtt{ba}$

ESP Dictionary	
Rule	string(·)
$\mathtt{A} \to \mathtt{aa}$	$\mathtt{a}^2$
$\mathtt{B} \to \mathtt{aaa}$	$\mathtt{a}^3$
$\mathtt{C} \to \mathtt{AA}$	$\mathtt{a}^4$
$\mathtt{D} \to \mathtt{BB}$	$\mathtt{a}^6$
$\mathtt{E} \to \mathtt{BBB}$	$\mathtt{a}^9$
$\mathtt{F} \to \mathtt{DD}$	$\mathtt{a}^{12}$
$\mathtt{G} \to \mathtt{NN}$	$(\mathtt{ba})^2$
$\mathtt{H} \to \mathtt{NNN}$	$(\mathtt{ba})^3$
$\mathtt{M} \to \mathtt{CG}$	$\mathtt{a}^4(\mathtt{ba})^2$
$\mathtt{U} \to \mathtt{ANN}$	$\mathtt{a}^2(\mathtt{ba})^2$
$\mathtt{R} \to \mathtt{JJJ}$	$(\mathtt{ab})^3$

HSP Dictionary	
Rule	string(·)
$\mathtt{a}_2 \to \mathtt{aa}$	$\mathtt{a}^2$
$\mathtt{a}_3 \to \mathtt{aaa}$	$\mathtt{a}^3$
$\mathtt{P} \to \mathtt{a}_3\mathtt{J}$	$\mathtt{a}^4\mathtt{b}$
$\mathtt{Q} \to \mathtt{a}_3\mathtt{I}$	$\mathtt{a}^5\mathtt{b}$

Fig. 4.6: Names of the ESP (Sect. 4.2.2) and HSP (Sect. 4.3) nodes stored in the global dictionary of our examples. The common dictionary contains all names that are used by both ESP and HSP. Each name occurs on the left side only once across all dictionaries.

contains a string of symbols with length two or three (this string is in $\Sigma_{h-1}^2 \cup \Sigma_{h-1}^3$). We maintain an injective dictionary $\mathfrak{D} : \Sigma_h \to \Sigma_{h-1}^2 \cup \Sigma_{h-1}^3$ to map a block to its symbols. The dictionary entries are of the form $\beta \to xy$ or $\beta \to xyz$, where $\beta \in \Sigma_h$ and $x, y, z \in \Sigma_{h-1}$. We write $\mathfrak{D}(X) := \mathfrak{D}(X[1]) \cdots \mathfrak{D}(X[|X|]) \in \Sigma_{h-1}^*$ for $X \in \Sigma_h^*$. Each block on height h is contained in a meta-block μ on height $h-1$, which is equal to a substring $\langle Y \rangle_{h-1}[i \mathinner{..} j] \in \Sigma_{h-1}^*$. We call the elements of $\langle Y \rangle_{h-1}[i \mathinner{..} j] \in \Sigma_{h-1}^*$ the *symbols* of μ. Since each application of esp reduces the string length by at least one half, there is an integer k with $k \leq \lg |Y|$ such that $\langle Y \rangle_k = \mathsf{esp}(\langle Y \rangle_{k-1})$ is a single block $\tau \in \Sigma_k$. We write $\mathcal{V} := \bigcup_{1 \leq h \leq k} \Sigma_h$ for the set of names in $\langle Y \rangle_1, \langle Y \rangle_2, \ldots, \langle Y \rangle_k$. The CFG for Y is represented by the non-terminals (i.e., the names) $\mathcal{V}$, the terminals Σ_0, the dictionary $\mathfrak{D}$, and the start symbol τ. This grammar exactly derives Y.

Throughout this chapter, we comply with the convention to write symbols in typewriter font, in particular, characters (elements of Σ_0) in lowercase and names (elements of Σ_h with $h \geq 1$) in uppercase letters. All examples use the same dictionary such that reappearing names are identical (see Fig. 4.6 for the used dictionary). Names restricted to a particular figure can be written with Greek letters (a necessity due to the limitation of having only 26 letters in the English alphabet).

The *ESP tree* $\mathsf{ET}(Y)$ of a string Y is the derivation tree of the CFG defined above. Its root node is the start symbol τ. The nodes on height h are $\langle Y \rangle_h$ for each height $h \geq 1$. In particular, the leaves are $\langle Y \rangle_1$. Each leaf refers to a substring in Σ_0^2 or Σ_0^3. The *generated substring* of a node $\langle Y \rangle_h[i]$ is the

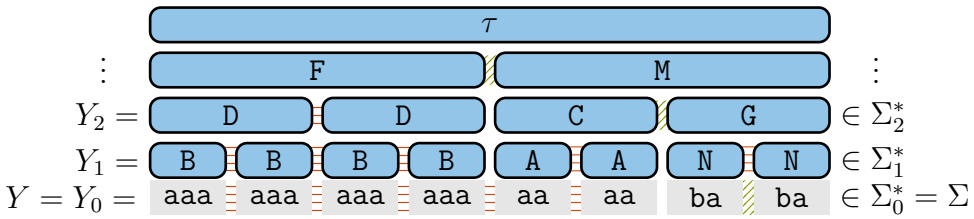


Fig. 4.7: The ESP tree of the string $Y =$ aaaaaaaaaaaaaaaababa. Like in Fig. 4.5, nodes belonging to the same meta-block are connected by red horizontal (▤) or green diagonal lines (▨) in case that they belong to a repeating or a Type 2 meta-block, respectively.

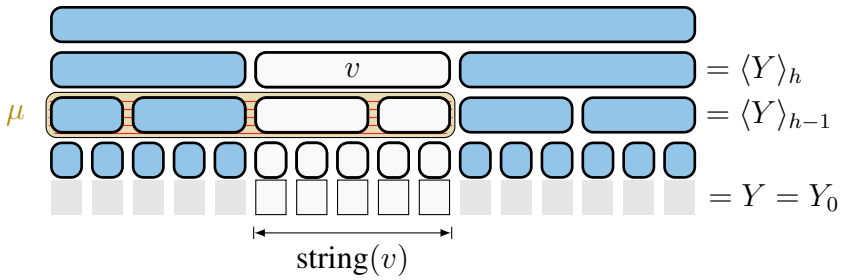


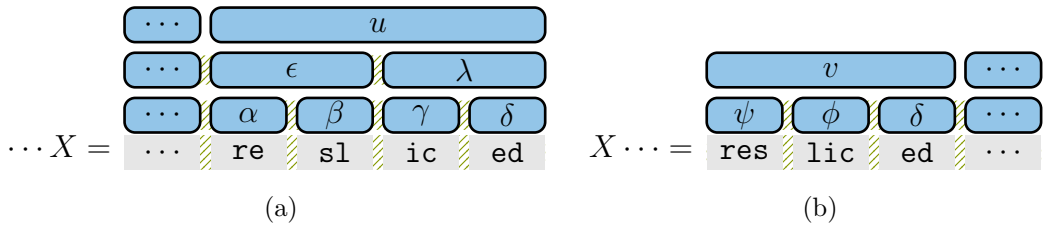
Fig. 4.8: $\langle Y \rangle_h$ with a highlighted node v. The subtree rooted at v is depicted by the white, rounded boxes. The generated substring $\mathsf{string}(v)$ of v is the concatenation of the white rectangular blocks on the lowest level in the picture. The meta-block μ, on which v is built, is the rounded golden (▢) rectangle covering the children of v and all nodes connected by a horizontal hatching (▤) on height $h - 1$.

substring of Y generated by the symbol $\langle Y \rangle_h[i]$ (applying the h-th iterate of $\mathfrak{D}$ to $\langle Y \rangle_h[i]$, yields a substring of Y, i.e., $\mathfrak{D}^{(h)}(\langle Y \rangle_h[i]) \in \Sigma^*$). We denote the generated substring of $\langle Y \rangle_h[i]$ by $\mathsf{string}(\langle Y \rangle_h[i])$. For instance, in Fig. 4.7, $\mathsf{string}(\mathtt{M}) = \mathtt{aaaababa}$. A node v on height h is said to be *built* on $\langle Y \rangle_{h-1}[b\,..\,e]$ if $\langle Y \rangle_{h-1}[b\,..\,e]$ contains the children of v. Like with blocks, nodes inherit the type of the meta-block on which they are built. An overview of the definitions is given in Fig. 4.8.

4.2.3.1 Shortcomings of ESP trees

In what follows, we present two shortcomings of the ESP trees. The first is that nodes with different names can have the same generated substring, i.e., $\mathfrak{D}^{(h)} : \Sigma_h \to \Sigma_0^*$ is not injective for $h \geq 2$ in general. The second is that it is not straight-forward to see which nodes of $\mathsf{ET}(Y)$ and $\mathsf{ET}(Z)$ are equal when Y is a substring of Z. Both cause problems when comparing subtrees of two nodes, which we later do for answering LCE queries.

Given two nodes u and v, it holds that $\mathsf{string}(u) = \mathsf{string}(v)$ if their names are



(a) (b)

Fig. 4.9: Excerpts of (a) $\mathsf{ET}(\cdots X)$ and (b) $\mathsf{ET}(X \cdots)$ with $X \coloneqq \texttt{resliced}$. Under the assumption that $\lg^* \sigma = 8$, the common substring X can be blocked differently in both trees (depending on the characters preceding X in the right figure).

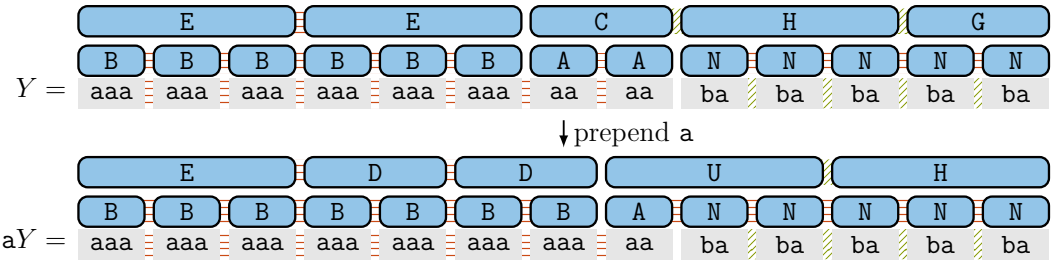


Fig. 4.10: Excerpt of $\mathsf{ET}(Y)$ and $\mathsf{ET}(\mathtt{a}Y)$ (higher nodes omitted), where $Y = \mathtt{a}^{9k+4}(\mathtt{ba})^{3k-1} = \mathtt{a}^{22}(\mathtt{ba})^5$ for $k = 2$. For all $k \geq 2$, there is a unique node in $\langle Y \rangle_2$ with the name C. This name does not appear in $\mathsf{ET}(\mathtt{a}Y)$.

equal. However, the other way around is not true in general. With $\mathsf{string}(u) = \mathsf{string}(v)$, it is not even assured that u and v are nodes sharing the same height. Suppose that Σ is a large alphabet with $\lg^* \sigma = 6$, and that $X \coloneqq \texttt{resliced}$ occurs in the text that we parse with ESP (see Fig. 4.9). We parse an occurrence of X either (a) with the alphabet reduction if it is within a Type 2 meta-block, or (b) greedily if it is at the beginning of a Type 2 meta-block. In the former case (a), we apply the alphabet reduction and end at a reduced alphabet with the characters $\{0, 1, 2\}$. Suppose that this occurrence of X is reduced to the string in superscript of $\cdots \overset{1\,0\,2\,1\,0\,1\,2\,1\,0\,1}{\texttt{resliced}} \cdots$. Then ESP creates the four blocks $\cdots |\texttt{re}|\texttt{sl}|\texttt{ic}|\texttt{ed}| \cdots$, whose boundaries are determined by the alphabet reduction. Further suppose that an application of esp creates two nodes of these blocks, which are put into a node u by an additional parse such that $\mathsf{string}(u) = X$. In the latter case (b), ESP creates the first two blocks of $\texttt{res}|\texttt{lic}|\texttt{ed}| \cdots$ greedily. Suppose that an additional parse puts these blocks in a node v such that $\mathsf{string}(v) = X$. Although $\mathsf{string}(v) = \mathsf{string}(u)$, the children of both nodes have different names, and therefore, both nodes cannot have the same name.

The second shortcoming is that it is not clear how to transfer the property of the alphabet reduction described in Lemma 4.5 from blocks to nodes. Given a substring Y of a string Z, the task is to analyze whether a node $\langle Y \rangle_h[i]$ in $\mathsf{ET}(Y)$

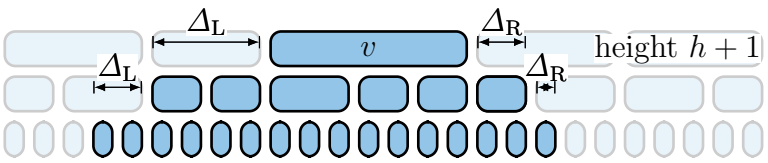


Fig. 4.11: Local surrounding of a node v at height $h+1$.

is also present in the tree $\mathsf{ET}(Z)$, i.e., we analyze changes of a node $\langle Y\rangle_h[i]$ when prepending or appending (pre-/appending) characters to Y. For the sake of analysis, we distinguish the two terminologies *block* and *node*, although a node is represented by a block: When we analyze a block in $\mathsf{esp}(X) \in \Sigma_h^*$ for a string $X \in \Sigma_{h-1}^*$, we let X to be subject to pre-/appending characters of Σ_{h-1}, whereas when we analyze a node $\langle Y\rangle_h[i]$ on a height h of $\mathsf{ET}(Y)$ of a string $Y \in \Sigma^*$, we let Y to be subject to pre-/appending characters of Σ. In this terminology, a block in $\mathsf{esp}(X)$ is only determined by X, whereas $\langle Y\rangle_h[i]$ is not only determined by $\mathsf{esp}^{(h-1)}(Y) \in \Sigma_{h-1}^*$, but also by Y itself. The difference is that a surrounded Type 2 block of $\mathsf{esp}(X)$ cannot be changed by pre-/appending characters to X due to Lemma 4.5, whereas we fail to find integers $\Delta_{L,h}$ and $\Delta_{R,h}$ such that a Type 2 node on height h built on $\langle Y\rangle_{h-1}[\Delta_{L,h} \mathinner{..} \Delta_{R,h}]$ cannot be changed by pre-/appending characters to Y. That is because the names inside $\langle Y\rangle_{h-1}$ and $\langle \mathtt{a}Y\rangle_{h-1}$ for $h \geq 2$ can differ at arbitrary positions. This can be seen in the following example: When parsing the string $Y := \mathtt{a}^{9k+4}(\mathtt{ba})^{3k-1}$ with the names defined in Fig. 4.6, we obtain $\mathsf{esp}(\mathsf{esp}(Y)) = \mathsf{esp}(\mathtt{B}^{3k}\mathtt{AAN}^{3k-1}) = \mathtt{E}^k\mathtt{CH}^{k-1}\mathtt{G}$. Let us focus on the unique occurrence of the name `C`, which is depicted in Fig. 4.10 for $k = 2$. On the one hand, there is a block in $\langle Y\rangle_1$ with the name `C` on height two. This block is surrounded for a sufficiently large k. Even for $k \geq 1$, it is easy to see that there is no way to change the name of this block by pre-/appending characters to the string $\mathtt{B}^{3k}\mathtt{AAN}^{3k-1}$. On the other hand, there is a unique node in $\mathsf{ET}(Y)$ with name `C` on height two. Regardless of the value of k, prepending `a` to Y changes the name of v: $\mathsf{esp}(\mathsf{esp}(\mathtt{a}Y)) = \mathsf{esp}(\mathtt{B}^{3k+1}\mathtt{AN}^{3k-1}) = \mathtt{E}^{k-1}\mathtt{DDUH}^{k-1}$.

In the following, we introduce the notion of surrounded nodes, since they are helpful to find rules that determine nodes that cannot be changed by pre-/appending characters.

4.2.3.2 Surrounded Nodes

Analogously to blocks, we classify nodes as surrounded when they are neighbored by sufficiently many nodes: A leaf is called *surrounded* if its generated substring is surrounded. The local surrounding of a leaf is the local surrounding of the block represented by the leaf. Given an internal node v on height $h+1$ ($h \geq 1$) whose children are $\langle Y\rangle_h[\beta]$, the *local surrounding* of v is the union of the nodes $\langle Y\rangle_h[\mathsf{b}(\beta) - \Delta_L \mathinner{..} \mathsf{e}(\beta) + \Delta_R]$ and the local surrounding of each node in $\langle Y\rangle_h[\mathsf{b}(\beta) - \Delta_L \mathinner{..} \mathsf{e}(\beta) + \Delta_R]$. If all nodes in the local surrounding of v are surrounded, we say that v is *surrounded*. Otherwise, we say that v is *non-surrounded*. See Fig. 4.11 for an illustration.

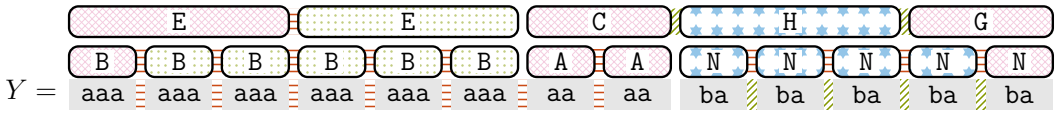
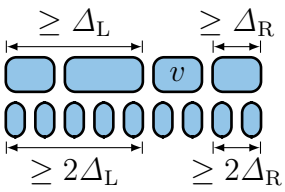


Fig. 4.12: $\mathsf{ET}(Y)$ of Fig. 4.10 with fragile, semi-stable and stable nodes highlighted. The fragile nodes are cross-hatched (▨), the semi-stable nodes are dotted (⬚), and the stable nodes have stars attached (✶). The leftmost nodes of the tree change their names when prepending one `b`. When prepending `a`'s, we observe that the children of the node with name `C` change. Assuming that $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ (and hence $|\Sigma| = 2$), only the rightmost node of the meta-block containing nodes with name `N` is fragile.

Lemma 4.6. There are at most $\Delta_\mathrm{L} + \Delta_\mathrm{R}$ many non-surrounded nodes on each height, summing up to $\mathcal{O}(\lg^* n \lg n)$ non-surrounded nodes in total.

Proof. We show the following claim: A node v on height h is surrounded if it has Δ_L preceding and Δ_R succeeding nodes. This is clear on height one by definition. Under the assumption that the claim holds for height $h-1$, v's preceding (resp. succeeding) nodes have at least $2\Delta_\mathrm{L}$ (resp. $2\Delta_\mathrm{R}$) children in total, where at least the Δ_L rightmost nodes (resp. Δ_R leftmost nodes) are surrounded by the assumption. Hence, v is surrounded. □



The examples of Sect. 4.2.3.1 shedding a light on the difference between blocks and nodes reveal that the property for surrounded blocks as shown on the right side of Fig. 4.4 cannot be transferred to surrounded nodes directly, since a surrounded node depends not only on its local surrounding, but also on the nodes on which it is built. Despite this discovery, we show that surrounded nodes can help us to create rules that are similar to Lemma 4.5.

4.2.4 Fragile and Stable Nodes in ESP Trees

We now analyze which nodes of $\mathsf{ET}(Y)$ are still present in $\mathsf{ET}(XYZ)$ for all strings X and Z. A node $\langle Y\rangle_h[j]$ in $\mathsf{ET}(Y)$ at a height h is said to be *stable* if, for *all* strings X and Z, there exists a node $\langle XYZ\rangle_h[k]$ in $\mathsf{ET}(XYZ)$ with the same name as $\langle Y\rangle_h[j]$ and $|X| + \sum_{i=1}^{j-1} |\mathsf{string}(\langle Y\rangle_h[i])| = \sum_{i=1}^{k-1} |\mathsf{string}(\langle XYZ\rangle_h[i])|$. We also consider repeating nodes that are present with slight *shifts*; a non-stable repeating node $\langle Y\rangle_h[j]$ in $\mathsf{ET}(Y)$ is said to be *semi-stable* if, for *all* strings X and Z, there exists a node $\langle XYZ\rangle_h[k]$ in $\mathsf{ET}(XYZ)$ with the same name as $\langle Y\rangle_h[j]$ and $\sum_{i=1}^{k-1} |\mathsf{string}(\langle XYZ\rangle_h[i])| - |S| < |X| + \sum_{i=1}^{j-1} |\mathsf{string}(\langle Y\rangle_h[i])| < \sum_{i=1}^{k-1} |\mathsf{string}(\langle XYZ\rangle_h[i])| + |S|$, where $S = \mathsf{string}(\langle Y\rangle_h[j]) = \mathsf{string}(\langle XYZ\rangle_h[k])$.

Nodes that are neither stable nor semi-stable are called *fragile*. By definition, the children of the (semi-)stable nodes (resp. fragile nodes) are also (semi-)stable

(resp. fragile). Figure 4.12 shows an example, where all three types of nodes are highlighted. The rest of this section studies how many fragile nodes exist in $\mathsf{ET}(Y)$.

As a warm-up, we first restrict the ESP tree construction on strings that are square-free. Since a name of the ESP tree is determined by its generating substring, $\mathsf{ET}(Y)$ cannot contain two consecutive occurrences of the same name on any height. We conclude that $\mathsf{ET}(Y)$ has no repeating nodes, i.e., it consists only of Type 2 nodes.

Remembering Sect. 4.2.2, the ESP parsing differs from the signature encoding in the auxiliary parsing used for the repeating meta-blocks and the first $\mathcal{O}(\lg^* n)$ symbols of a Type 2 meta-block. The signature encoding introduces an intermediate step where it replaces all runs with smallest period one with a new symbol such that no symbol occurs at two adjacent positions in the resulting string. This means that the signature encoding can apply the alphabet reduction on the *entire* string after applying this intermediate step. By doing so, the signature encoding introduces at most $\mathcal{O}(\lg^* n)$ fragile nodes on each height [198, Lemma 9]. In the case of a square-free string, the auxiliary parsing is only required for the $\mathcal{O}(\lg^* n)$ leftmost symbols on each height of both parsings (signature encoding and ESP): (a) the intermediate step of the signature encoding does not introduce any new symbols, and (b) the ESP creates only a single Type 2 meta-block. Hence in this case, the maximal numbers of fragile nodes (a) in the signature encoding parse trees and (b) in the ESP tree have the same asymptotic upper bound. For completeness, we prove this statement explicitly, as the techniques will be used later to devise an upper bound in the general case. We start with the following lemma:

Lemma 4.7 ([55, Lemma 8])**.** A Type 2 node is stable if (a) it is surrounded and (b) its local surrounding does not contain a fragile node.

With Lemma 4.7 we immediately obtain:

Lemma 4.8. Given a square-free string Y, a fragile node of $\mathsf{ET}(Y)$ is a non-surrounded node.

Proof. According to Lemma 4.7, we can bound the number of fragile nodes by the number of those nodes that do not satisfy the conditions in Lemma 4.7. Since $\mathsf{ET}(Y)$ only contains Type 2 nodes, we can inductively show that a fragile node is non-surrounded for all heights of the ESP tree: Surrounded leaves are stable due to Lemma 4.5. Therefore, the claim holds for $h = 1$. By definition, a node v on height h is surrounded if its local surrounding S on height $h - 1$ is surrounded. Given that the claim holds for $h - 1$, a node in S can only be fragile if it is not surrounded. This concludes that v can be fragile only if it is not surrounded. □

Combining Lemma 4.8 with Lemma 4.6 yields the following corollary:

Corollary 4.9. The number of fragile nodes of an ESP tree built on a square-free string of length n is $\mathcal{O}(\lg^* n \lg n)$. On each height, it contains $\mathcal{O}(\lg^* n)$ fragile nodes.

In the following we present a lower and an upper bound on the number of fragile nodes. First, we show that the ESP technique can change $\Omega(\lg^2 n)$ nodes when changing a single character of the input string. The idea is to give an example that contains a large number of Type M meta-blocks in a specific constellation. Remembering how the ESP technique parses its input, a remaining single symbol neighbored by two repeating meta-blocks is fused with one of them to form a Type M meta-block. We provide examples for Rule (M) and for the original tie breaking rule for Type M meta-blocks:

> Rule (M'): Fuse a remaining character $Y[i]$ with its *preceding* meta-block, or, if $i = 1$, with its succeeding meta-block.

Each example presents a string of length at most n whose ESP tree has $\Omega(\lg^2 n)$ fragile nodes. These examples contradict Lemma 9 in [55], where it is claimed that there are $\mathcal{O}(\lg^* n \lg n)$ fragile nodes in the ESP tree of a text of length n.

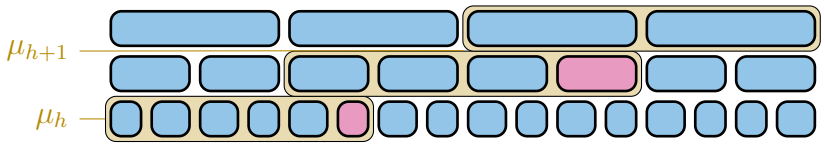


Fig. 4.13: Basic idea of our two counter examples described in Thms. 4.11 and 4.13. We build a counter example in such a way that the last node of a certain repeating meta-block μ_h on height h (a) is fragile (■), and (b) is the child of the first node of a repeating meta-block μ_{h+1} having the same properties as μ_h. This property can cause a recursive chain reaction when prepending a suitable character such that the names of the last and the first block of a meta-block on height h and a meta-block on height $h + 1$ are changed, on each height h.

4.2.4.1 Fusing with the preceding repeating meta-block Rule (M')

Consider a Type 1 meta-block μ whose rightmost node is fragile. If the leftmost node of a repeating meta-block ν is built on μ's rightmost node, then the rightmost node of ν can also be fragile.

Having this idea in mind, we build an example consisting of a chain of repeating meta-blocks, where the leftmost node of a repeating meta-block is built on the fragile rightmost node of a meta-block of one depth below (Fig. 4.13). The main idea is the following: Each meta-block of this chain can be of arbitrary (but sufficiently long) length. Keeping in mind that changing the name of a

node means that the names of its ancestors also have to change, we can create an example string whose ESP tree contains fragile nodes appearing on each height at arbitrary positions. Before giving such an example we introduce a lemma showing the associativity of esp on a special class of strings, which helps us proving the lower bound:

Lemma 4.10. Suppose that we comply to Rule (M'). Given a height h and two strings X, Y that are either empty or have a length of at least $2 \cdot 3^{h-1}$, $\mathsf{esp}^{(h)}(X\mathtt{b}^{3^i}Y) = \mathsf{esp}^{(h)}(X)\,\mathsf{esp}^{(h)}(\mathtt{b}^{3^i})\,\mathsf{esp}^{(h)}(Y)$ if $i \geq h$, $\mathtt{b}$ is neither a suffix of X nor a prefix of Y, and there is no prefix of $\mathsf{esp}^{(j)}(Y)$ of the form $\mathtt{cd}^k$ for some symbols $\mathtt{c}, \mathtt{d} \in \Sigma_j$ with $\mathtt{c} \neq \mathtt{d}$, and integers k, j with $k \geq 2$ and $0 \leq j \leq h-1$.

Proof. The additional requirement for Y is to ensure that the leftmost block of $\mathsf{esp}^{(j)}(Y)$ is not a non-repetitive Type M block that has been fused to its succeeding meta-block, only because it has no preceding meta-block. Regardless of which symbols are prepended to $\mathsf{esp}^{(j-1)}(Y)$, the first symbol of such a block would form with its preceding symbols a new block.

For $h = 1$, esp divides the string $X\mathtt{b}^{3^i}Y$ into meta-blocks such that there is one Type 1 meta-block μ that exactly contains the substring $\mathtt{b}^{3^i}$. That is because of the following: If X (resp. Y) is not the empty string, then X (resp. Y) contains at least two characters. Since we favor fusing with the preceding meta-block, there is no chance that characters of X can enter μ. Assume that Y is not the empty string. Since the first block of $\mathsf{esp}(Y)$ is neither a non-repetitive Type M block nor a block starting with $\mathtt{b}$, it is not possible that characters of this block can enter μ.

Under the assumption that the claim holds for a given $h - 1 \geq 0$, we have

$$\begin{aligned}\mathsf{esp}^{(h)}(X\mathtt{b}^{3^i}Y) &= \mathsf{esp}\Big(\mathsf{esp}^{(h-1)}(X\mathtt{b}^{3^i}Y)\Big)\\ &= \mathsf{esp}\Big(\mathsf{esp}^{(h-1)}(X)\,\mathsf{esp}^{(h-1)}(\mathtt{b}^{3^i})\,\mathsf{esp}^{(h-1)}(Y)\Big).\end{aligned}$$

The strings $\mathsf{esp}^{(h)}(X)$ and $\mathsf{esp}^{(h)}(Y)$ are either empty or contain at least two symbols. Since $i \geq h$, $\mathsf{esp}^{(h-1)}(\mathtt{b}^{3^i})$ is the repetition of the same symbol. This repetition has a length of at least three such that we can apply the shown associativity for $h = 1$ to show the claim. □

Theorem 4.11. There is a text of length n whose ESP tree has $\Omega(\lg^2 n)$ fragile nodes when complying to Rule (M').

Proof. Let $\mathtt{a}$, $\mathtt{b}$, and $\mathtt{c} \in \Sigma$ be three different characters. In the following, we show that the text

$$Y \coloneqq (X_0)^{3^k}(X_1)^{3^{k-1}}(X_2)^{3^{k-2}} \cdots (X_{k-1})^3$$

with $k \coloneqq \lfloor \log_3(n/\log_3 n) \rfloor$ has a length at most n, and its ESP tree has $\Omega(\lg^2 n)$ fragile nodes, where

$$X_0 \coloneqq \mathtt{a}, \text{ and } X_i \coloneqq \begin{cases} X_{i-1}^2\mathtt{b}^{3^{i-1}} & \text{if } i \text{ is odd,} \\ X_{i-1}^2\mathtt{c}^{3^{i-1}} & \text{if } i \text{ is even,} \end{cases} \text{ for } i = 1, \ldots, k.$$

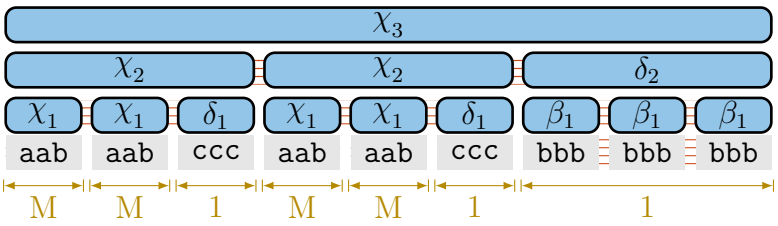


Fig. 4.14: $\mathsf{ET}(X_3)$ as defined in Thm. 4.11. The subtree of each node with name χ_i is equal to $\mathsf{ET}(X_i)$. The meta-blocks of the lowest height are labeled with their types.

For instance, $X_0 = \mathtt{a}$, $X_1 = \mathtt{aab}$, $X_2 = \mathtt{aabaabc}^3$, and $X_3 = X_2^2\mathtt{b}^9$.

We start with determining the length of Y. Since $|X_0| = 3^0$, under the assumption that $|X_i| = 3^i$, we obtain that $|X_{i+1}| = 2\,|X_i| + 3^i = 3^{i+1}$. Therefore, $\left|X_i^{3^{k-i}}\right| = 3^k$ for all $i = 0, \ldots, k-1$. We conclude that the length of Y is at most n, since $|Y| = k3^k \leq n \log_3(n/\log_3 n)/\log_3 n \leq n$.

We now show that each substring X_i of Y is the generated substring of a node χ_i of $\mathsf{ET}(Y)$ on height i whose subtree is equal to the perfect ternary subtree $T_i := \mathsf{ET}(X_i)$, for $i = 1, \ldots, k-1$. This is true for $i = 1, 2, 3$, as can be seen in Fig. 4.14. For the general case, we adapt the associativity shown for esp in Lemma 4.10 to the string X_i:

Sub-Claim. For every i with $0 \leq i \leq k-2$ it holds that

(I) $\left|\mathsf{esp}^{(i+1)}(X_{i+1})\right| = 1$,

(II) $$\begin{aligned}\mathsf{esp}^{(h)}(X_{i+1}) &= \mathsf{esp}^{(h)}(X_iX_i\mathtt{d}_i^{3^i}) \\ &= \mathsf{esp}^{(h)}(X_iX_i)\,\mathsf{esp}^{(h)}(\mathtt{d}_i^{3^i}) \\ &= \mathsf{esp}^{(h)}(X_i)\,\mathsf{esp}^{(h)}(X_i)\,\mathsf{esp}^{(h)}(\mathtt{d}_i^{3^i}),\text{ and}\end{aligned}$$

(III) $\mathsf{esp}^{(h)}(X_{i+1})$ starts with a repetition of a symbol,

for every h with $0 \leq h \leq i$, where $\mathtt{d}_i$ is a character with $\mathtt{d}_i = \mathtt{b}$ if i is even, otherwise $\mathtt{d}_i = \mathtt{c}$.

Sub-Proof. For $i = 0$ we have

(I) $\left|\mathsf{esp}^{(1)}(X_1)\right| = |\mathsf{esp}(\mathtt{aab})| = 1$ ($\mathtt{aab}$ is put in a Type M meta-block having exactly one block),

(II) $\mathsf{esp}^{(0)}(X_1) = X_1$, and

(III) $X_1 = \mathtt{aab}$ starts with a repetition of the character $\mathtt{a}$.

Under the assumption that the claim holds for an integer i, we conclude that it holds for $i+1$ due to

$$\begin{aligned}
\mathrm{esp}^{(h)}(X_{i+2}) =& \mathrm{esp}^{(h)}\left(X_{i+1}X_{i+1}\mathtt{d}_{i+1}^{3^{i+1}}\right)\\
=& \mathrm{esp}^{(h)}\left(X_iX_i\mathtt{d}_i^{3^i}X_iX_i\mathtt{d}_i^{3^i}\mathtt{d}_{i+1}^{3^{i+1}}\right)\\
\text{(Lemma 4.10,}\mathtt{d}_i\neq\mathtt{d}_{i+1}\text{)} =& \mathrm{esp}^{(h)}\left(X_iX_i\mathtt{d}_i^{3^i}X_iX_i\mathtt{d}_i^{3^i}\right)\mathrm{esp}^{(h)}\left(\mathtt{d}_{i+1}^{3^{i+1}}\right)\\
\text{(Lemma 4.10,(I) or (III))} =& \mathrm{esp}^{(h)}(X_iX_i)\mathrm{esp}^{(h)}\left(\mathtt{d}_i^{3^i}\right)\mathrm{esp}^{(h)}(X_iX_i)\mathrm{esp}^{(h)}\left(\mathtt{d}_i^{3^i}\right)\\
& \mathrm{esp}^{(h)}\left(\mathtt{d}_{i+1}^{3^{i+1}}\right)\\
\text{(Lemma 4.10,(I) or (III))} =& \mathrm{esp}^{(h)}\left(X_iX_i\mathtt{d}_i^{3^i}\right)\mathrm{esp}^{(h)}\left(X_iX_i\mathtt{d}_i^{3^i}\right)\mathrm{esp}^{(h)}\left(\mathtt{d}_{i+1}^{3^{i+1}}\right)\\
=& \mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}\left(\mathtt{d}_{i+1}^{3^{i+1}}\right)
\end{aligned}$$

for $1 \le h \le i$. The conditions of Lemma 4.10 hold because $\mathtt{d}_i$ is neither a prefix nor a suffix of X_i, $\mathtt{d}_i \neq \mathtt{d}_{i+1}$, $|X_iX_i| = 2\cdot 3^i$, and $\mathrm{esp}^{(h)}(X_iX_i)$ starts with a repetition of a symbol due to

$$\begin{cases}
\text{(III)} & \text{for } h < i, \text{ or due to}\\
\mathrm{esp}^{(i)}(X_iX_i) =^{(II)} \mathrm{esp}^{(i)}(X_i)\mathrm{esp}^{(i)}(X_i) \text{ and (I)} & \text{for } h = i.
\end{cases}$$

For $h = i+1$ we use that (I) holds for X_i, $\left|\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\right| = 1$, and $\mathrm{esp}^{(i)}(\mathtt{d}_{i+1}^{3^{i+1}})$ is a repetition of length 3 of the same symbol, to obtain

$$\begin{aligned}
\mathrm{esp}^{(i+1)}(X_{i+2}) =& \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_{i+2})\right)\\
=& \mathrm{esp}\Big(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\\
& \mathrm{esp}^{(i)}(\mathtt{d}_{i+1}^{3^{i+1}})\Big)\\
\text{(Lemma 4.10)} =& \mathrm{esp}\Big(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\Big)\\
& \mathrm{esp}\left(\mathrm{esp}^{(i)}(\mathtt{d}_{i+1}^{3^{i+1}})\right)\\
\text{(evaluate and reformulate)} =& \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\right)\\
& \mathrm{esp}\left(\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\right)\mathrm{esp}\left(\mathrm{esp}^{(i)}(\mathtt{d}_{i+1}^{3^{i+1}})\right),
\end{aligned}$$

where we used that another application of esp puts $\mathrm{esp}^{(i)}(X_iX_i)\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})$ into a single Type M meta-block of length three, and that $\mathtt{d}_i$ is neither a prefix nor a suffix of X_i. This concludes (II). A consequence is (III): For $h \le i$ we have $\mathrm{esp}^{(h)}(X_{i+2}) = \mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}(X_{i+1})\mathrm{esp}^{(h)}(\mathtt{d}_{i+1}^{3^{i+1}})$, and $\mathrm{esp}^{(h)}(X_{i+1})$ starts with a repetition of a symbol according to our assumption. For $h = i+1$ we have

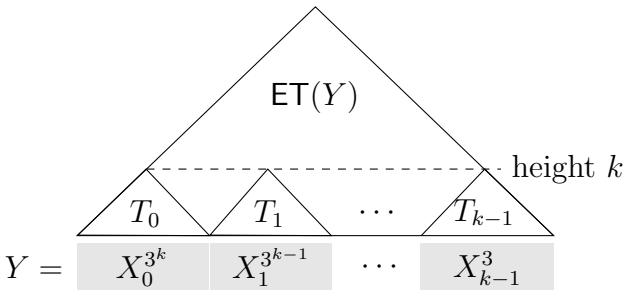
$$\begin{aligned}
\mathrm{esp}^{(i+1)}(X_{i+2}) = \mathrm{esp}\Big(&\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\\
&\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}(X_i)\,\mathrm{esp}^{(i)}(\mathtt{d}_i^{3^i})\,\mathrm{esp}^{(i)}(\mathtt{d}_{i+1}^{3^{i+1}})\Big).
\end{aligned}$$

Due to (I), $\left|\mathsf{esp}^{(i)}(X_i)\right| = \left|\mathsf{esp}^{(i)}(\mathsf{d}_i^{3^i})\right| = 1$; hence the last application of esp creates three blocks, where each of the first two represents the string $\mathsf{esp}^{(i)}(X_i)\,\mathsf{esp}^{(i)}(X_i)\,\mathsf{esp}^{(i)}(\mathsf{d}_i^{3^i})$ of length three. Another application of esp yields (I). ■

Let β_i and γ_i denote the names of the roots of $\mathsf{ET}(\mathsf{b}^{3^i})$ and of $\mathsf{ET}(\mathsf{c}^{3^i})$, respectively. Set $\delta_i \coloneqq \beta_i$ if i is even, otherwise $\delta_i \coloneqq \gamma_i$. Then $\langle X_{i+1}\rangle_{i+1} = \chi_{i+1}$ due to Sub-Claim (I), and $\langle X_{i+1}\rangle_i = \chi_i\chi_i\delta_i$ due to Sub-Claim (II). Consequently,

$$\mathsf{esp}\Big(\big(\langle X_{i+1}\rangle_i\big)^{3^{k-i-1}}\Big) = \mathsf{esp}\Big(\big(\chi_i\chi_i\delta_i\big)^{3^{k-i-1}}\Big) = \big(\mathsf{esp}(\chi_i\chi_i\delta_i)\big)^{3^{k-i-1}} = \chi_{i+1}^{3^{k-i-1}}. \tag{4.1}$$

This means that $\langle X_i\rangle_h^{3^{k-i}} = \langle X_i^{3^{k-i}}\rangle_h$ is a repetition of length 3^{k-h} consisting of the same name, for every height $h = i, \dots, k$. We conclude that $T_i \coloneqq \mathsf{ET}((X_i)^{3^{k-i}})$ is a perfect ternary tree.



Finally, we show that $\mathsf{esp}^{(h)}(Y) = \mathsf{esp}^{(h)}(X_1^{3^k}) \cdots \mathsf{esp}^{(h)}(X_{k-1}^{3^1})$ holds for each height h with $1 \le h \le k$. On the one hand, we have

$$\begin{aligned}
\mathsf{esp}^{(h)}\Big(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}\Big) &= \mathsf{esp}^{(h)}\Big(X_i^{3^{k-i-1}} X_{i-1} X_{i-1} \mathsf{d}_{i-1}^{3^{i-1}} X_{i+1}^{3^{k-i-1}}\Big) \\
\text{(III) with } 0\le i\le h-2 \quad &= \mathsf{esp}^{(h)}\Big(X_i^{3^{k-i-1}} X_{i-1} X_{i-1}\Big)\mathsf{esp}^{(h)}\Big(\mathsf{d}_{i-1}^{3^{i-1}}\Big) \\
&\qquad \mathsf{esp}^{(h)}\Big(X_{i+1}^{3^{k-i-1}}\Big) \\
&= \mathsf{esp}^{(h)}\Big(X_i^{3^{k-i-1}} X_{i-1} X_{i-1} \mathsf{d}_{i-1}^{3^{i-1}}\Big)\mathsf{esp}^{(h)}\Big(X_{i+1}^{3^{k-i-1}}\Big) \\
&= \mathsf{esp}^{(h)}\Big(X_i^{3^{k-i}}\Big)\mathsf{esp}^{(h)}\Big(X_{i+1}^{3^{k-i-1}}\Big)
\end{aligned} \tag{4.2}$$

for $1 \le h \le i-1$ due to Lemma 4.10. On the other hand, we have

$$\begin{aligned}
\mathsf{esp}^{(h)}\Big(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}\Big) &= \mathsf{esp}^{(h-i+1)}\Big(\mathsf{esp}^{(i-1)}\Big(X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}\Big)\Big) \\
\text{Eq. (4.2)} \quad &= \mathsf{esp}^{(h-i+1)}\Big(\mathsf{esp}^{(i-1)}\Big(X_i^{3^{k-i}}\Big)\,\mathsf{esp}^{(i-1)}\Big(X_{i+1}^{3^{k-i-1}}\Big)\Big) \\
\text{Eq. (4.1)} \quad &= \mathsf{esp}^{(h-i)}\Big(\mathsf{esp}\Big(\big(\chi_{i-1}\chi_{i-1}\delta_{i-1}\big)^{3^{k-i}} \\
&\qquad \big(\chi_{i-1}\chi_{i-1}\delta_{i-1}\chi_{i-1}\chi_{i-1}\delta_{i-1}\langle \mathsf{d}_i^{3^i}\rangle_{i-1}\big)^{3^{k-i-1}}\Big)\Big) \\
\text{(apply esp)} \quad &= \mathsf{esp}^{(h-i)}\Big(\chi_i^{3^{k-i}}\big(\chi_i\chi_i\delta_i\big)^{3^{k-i-1}}\Big) \\
&= \mathsf{esp}^{(h-i-1)}\Big(\mathsf{esp}\Big(\chi_i^{3^{k-i}}\chi_i\chi_i\delta_i\Big)\mathsf{esp}\Big(\big(\chi_i\chi_i\delta_i\big)^{3^{k-i-2}}\Big)\Big) \\
\text{(evaluate and reformulate)} \quad &= \mathsf{esp}^{(h-i-1)}\Big(\mathsf{esp}\Big(\chi_i^{3^{k-i}}\Big)\mathsf{esp}\Big(\big(\chi_i\chi_i\delta_i\big)^{3^{k-i-1}}\Big)\Big) \\
\text{Eq. (4.1)} \quad &= \mathsf{esp}^{(h-i-1)}\Big(\mathsf{esp}\Big(\chi_i^{3^{k-i}}\Big)\mathsf{esp}\Big(\chi_{i+1}^{3^{k-i-1}}\Big)\Big) \\
\text{(Lemma 4.10)} \quad &= \mathsf{esp}^{(h-i)}\Big(\chi_i^{3^{k-i}}\Big)\mathsf{esp}^{(h-i)}\Big(\chi_{i+1}^{3^{k-i-1}}\Big)
\end{aligned} \tag{4.3}$$

for $i \le h \le k$. It is easy to extend the pairwise associativity $X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}$ for each i with $0 \le i \le k-2$ to $X_1^{3^k} \cdots X_{k-1}^{3^1}$. This concludes that the root of T_i has the same name as the i-th leftmost node of $\mathsf{ET}(Y)$ on height k. Figure 4.15(left) shows an excerpt of T_i and T_{i+1}. The crucial step in Eq. (4.3) is the re-formulation of the parsing

$$\begin{aligned} &\mathsf{esp}^{(h-i-1)}(\underbrace{\mathsf{esp}(\chi_i^{3^{k-i}}\chi_i\chi_i\delta_i)}_{\text{belongs to } T_i}\underbrace{\mathsf{esp}((\chi_i\chi_i\delta_i)^{3^{k-i-2}})}_{\text{belongs to } T_{i+1}}) \\ &= \mathsf{esp}^{(h-i-1)}(\mathsf{esp}(\chi_i^{3^{k-i}})\underbrace{\mathsf{esp}((\chi_i\chi_i\delta_i)^{3^{k-i-1}})}_{=:\mu_{i+1}}) \end{aligned} \tag{4.4}$$

showing that there is a Type 1 meta-block μ_{i+1} covering all nodes of T_{i+1} and the rightmost node of T_i, on height $i+1$. This meta-block is a repetition of the symbol $\mathsf{esp}(\chi_i\chi_i\delta_i) = \chi_{i+1} \in \Sigma_{h+1}$.

Given that μ_0 is the first Type 1 meta-block of $\mathsf{esp}(Y)$ (covering the prefix $X_0^{3^h+2}$), we now examine what happens with μ_i for each i with $0 \le i \le h-1$ when removing the first $\mathtt{a}$ from Y. Let us call the shortened string Y', i.e., $Y = \mathtt{a}Y'$. On removing the first $\mathtt{a}$ from Y, we claim that the meta-block μ_i contains one symbol χ_i less, for every i with $0 \le i \le h-1$ (cf. Fig. 4.15 showing the difference between $\langle Y\rangle_i$ and $\langle Y'\rangle_i$ on height i with $0 \le i \le k-1$): For μ_0, this is trivial. For an $i \ge 0$, focus on the substring $X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}$ of Y: We have

$$\begin{aligned} \mathsf{esp}\Big(\langle X_i^{3^{k-i}} X_{i+1}^{3^{k-i-1}}\rangle_i\Big) &= \mathsf{esp}\Big(\chi_i^{3^{k-i}}(\chi_i\chi_i\delta_i)^{3^{k-i-1}}\Big) \\ &= \mathsf{esp}(\underbrace{\chi_i^{3^{k-i}}}_{\text{suffix of } \mu_i})\underbrace{\mathsf{esp}\Big((\chi_i\chi_i\delta_i)^{3^{k-i-1}}\Big)}_{=\mu_{i+1}} \\ &= \mathsf{esp}\Big(\chi_i^{3^{k-i}}\Big)\chi_{i+1}^{3^{k-i-1}} \end{aligned}$$

due to Eq. (4.3). Under the assumption that removing the first character $\mathtt{a}$ from Y causes μ_i to shrink by one symbol $\chi_i \in \Sigma_i$, we get

$$\begin{aligned} \mathsf{esp}\Big(\chi_i^{3^{k-i}-1}(\chi_i\chi_i\delta_i)^{3^{k-i-1}}\Big) &= \mathsf{esp}\Big(\chi_i^{3^{k-i}}\chi_i\delta_i\Big)\mathsf{esp}\Big((\chi_i\chi_i\delta_i)^{3^{k-i-1}-1}\Big) \\ &= \mathsf{esp}\Big(\chi_i^{3^{k-i}}\chi_i\delta_i\Big)\chi_{i+1}^{3^{k-i-1}-1} \\ &\neq \mathsf{esp}\Big(\chi_i^{3^{k-i-1}}\Big)\chi_{i+1}^{3^{k-i-1}}. \end{aligned}$$

We observe that the length of μ_i is decremented by one, causing the name of its rightmost block to change, which is the leftmost node of T_{i+1} on height $i+1$, and the first symbol of μ_{i+1}. Due to the tie breaking rule, this block gets fused with its preceding meta-block at height $i+1$, decrementing the length of its succeeding meta-block μ_{i+1} by one (and hence, this process repeats for all $i = 0, \ldots, k-2$). This means that the leftmost node on height i of T_i changes, for $1 \le i \le k-1$. Each of these nodes receives a new name such that it is fused with its preceding Type 1 meta-block to form a Type M meta-block. Since changing a node on

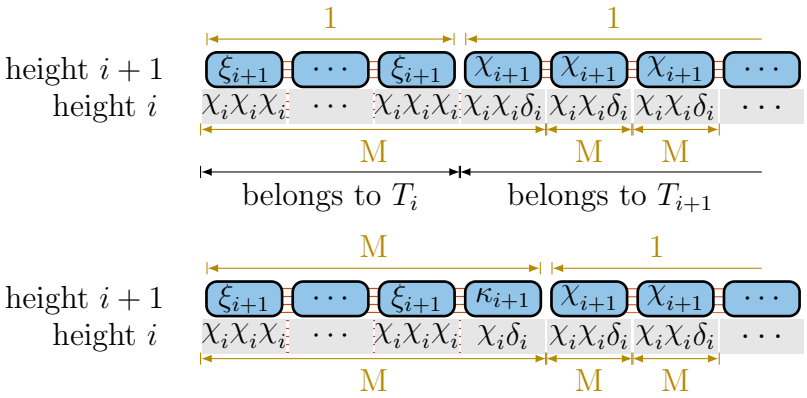


Fig. 4.15: Differences between $\mathsf{ET}(Y)$ (*top*) and $\mathsf{ET}(Y')$ (*bottom*) on the heights i and $i+1$, where $Y = \mathtt{a}^{3^k}(\mathtt{a}^2\mathtt{b})^{3^{k-1}}((\mathtt{a}^2\mathtt{b})^2\mathtt{c}^3)^{3^{k-2}}\cdots$ and $Y = \mathtt{a}Y'$ (defined in Thm. 4.11). The names ξ_{i+1} and κ_{i+1} are only used in this figure. The meta-blocks on height i and $i+1$ are labeled with their types.

height i changes all its ancestors (or removing the first character of Y for $i = 0$ changes all nodes built on this character), at least $k - i$ nodes are changed in T_i. In total, at least $k + (k-1) + (k-2) + \cdots + 2 = (k^2+k)/2 - 1$ nodes are changed. Hence, there is a lower bound of $\Omega(k^2) = \Omega(\log_3^2(n/\log_3 n)) = \Omega(\lg^2 n)$ fragile nodes. □

Note that the later introduced HSP technique (see Sect. 4.3) with the same tie breaking rule also produces $\Omega(\lg^2 n)$ fragile nodes in this example. However, this is not the case when complying with Rule (M), as we will see later in Sect. 4.3.1.

4.2.4.2 Fusing with the succeeding repeating meta-block

The idea is similar to the previous example. In particular, we introduce a corollary of Lemma 4.10:

Corollary 4.12. Given a height h and a string Y that is either empty or has a length of at least $2 \cdot 3^{h-1}$, $\mathsf{esp}^{(h)}(XY) = \mathsf{esp}^{(h)}(X)\,\mathsf{esp}^{(h)}(Y)$ if $\mathtt{a}$ is not a prefix of Y, where $X = \mathtt{b}^{3^i}\mathtt{a}^{3^j}$ with $i + j \geq h$, and $\mathtt{a}, \mathtt{b} \in \Sigma$ with $\mathtt{a} \neq \mathtt{b}$.

In the following example, we build a text whose ESP tree has a specific Type M meta-block on each height that we want to change. Given a Type M meta-block μ that emerged from prepending a symbol to a Type 1 meta-block, we can create a new meta-block by prepending another symbol such that it precedes μ and absorbs μ's first symbol (μ then returns to be a Type 1 meta-block). We can arrange the Type M meta-blocks such that prepending a symbol to the text changes a Type M meta-block on each height:

Theorem 4.13. There is a text of length n whose ESP tree has $\Omega(\lg^2 n)$ fragile nodes when complying with Rule (M).

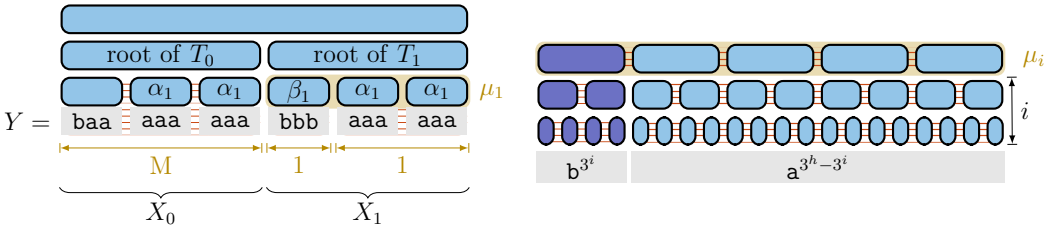


Fig. 4.16: $\mathsf{ET}(Y)$ of the example string Y defined in Thm. 4.13 with $k = 1$ (*left*) and as a schematic illustration (*right*) with the meta-block μ_i on height i (due to space issues the number of nodes/children is incorrect). The meta-blocks of the lowest height in the *left* figure are labeled with their types.

Proof. Let $k = \lfloor \log_3(n/\log_3 n) \rfloor$ be a natural number, and $\mathtt{a}, \mathtt{b} \in \Sigma$. Define

$$Y \coloneqq X_0 X_1 \cdots X_k \text{ with } X_i \coloneqq \mathtt{b}^{3^i}\mathtt{a}^{3^k-3^i},$$

for $0 \le i \le k-1$. Figure 4.16 gives an example on its left side. In the following, we show that $|Y| \le n$, and $\mathsf{ET}(Y)$ has $\Omega(\lg^2 n)$ fragile nodes.

Given an integer i with $0 \le i \le k-1$, we have $|X_i| = 3^k$ and $|Y| = k3^k \le n$. Corollary 4.12 yields $\mathsf{esp}^{(h)}(X_i) = \mathsf{esp}^{(h)}(\mathtt{b}^{3^i})\,\mathsf{esp}^{(h)}(\mathtt{a}^{3^k-3^i})$ for all heights h with $0 \le h \le i$, since $3^k - 3^i \ge 3^k - 3^{k-1} = 2 \cdot 3^{k-1}$. Let $\alpha_i \coloneqq \langle \mathtt{a}^{3^k} \rangle_i[1]$ and $\beta_i \coloneqq \langle \mathtt{b}^{3^k} \rangle_i[1]$ be the nodes on height i with $0 \le i \le k$ and, respectively, $\mathsf{string}(\alpha_i) = \mathtt{a}^{3^i}$ and $\mathsf{string}(\beta_i) = \mathtt{b}^{3^i}$ ($\alpha_0 \coloneqq \mathtt{a}$, $\beta_0 \coloneqq \mathtt{b}$).

The function esp applied on $\mathsf{esp}^{(h-1)}(X_i)$ partitions its input $\mathsf{esp}^{(h-1)}(X_i)$ into two meta-blocks: a Type 1 meta-block containing all β_i's, and a subsequent Type 1 meta-block containing all α_i's. All blocks of these two meta-blocks contain three symbols, since each meta-block has a length that is equal to a power of three. For the upper heights we get

$$\mathsf{esp}^{(h+i)}(X_i) = \mathsf{esp}^{(h)}\left(\overbrace{\underbrace{\mathsf{esp}^{(i)}\left(\mathtt{b}^{3^i}\right)}_{=\beta_i}\underbrace{\mathsf{esp}^{(i)}\left(\mathtt{a}^{3^k-3^i}\right)}_{=\alpha_i^{3^{k-i}-1}}}^{\text{having length } 3^{k-i}}\right) \quad \text{for } 0 \le h+i \le k-1. \tag{4.5}$$

Hence, $\mathsf{esp}^{(h+i)}(X_i)$ consists of exactly one Type M meta-block, which has length 3^{k-h-i}, and each block contains three symbols. We conclude that the tree $T_i \coloneqq \mathsf{ET}(X_i)$ is a perfect ternary tree, for $0 \le i \le k-1$. Since $\left|\mathsf{esp}^{(h)}(X_i)\right| = 3^{k-h}$ for all i, h with $0 \le i \le k-1$ and $0 \le h \le k$, with Cor. 4.12 it is easy to see that $\mathsf{esp}^{(h)}(Y) = \mathsf{esp}^{(h)}(X_1 \cdots X_{k-1}) = \mathsf{esp}^{(h)}(X_1) \cdots \mathsf{esp}^{(h)}(X_{k-1})$ for all $0 \le h \le k$. Consequently, X_i is the generated substring of the i-th leftmost node v_i of $\mathsf{ET}(Y)$ on height k. The name of v_i is the name of the root of T_i, for $0 \le i \le k-1$.

For the proof, we prepend an $\mathtt{a}$ to Y and call the new string Y', i.e., $Y' = \mathtt{a}Y$. Our analysis of the difference between $\mathsf{ET}(Y)$ and $\mathsf{ET}(Y')$ focuses on the unique meta-block at height i of T_i: From Eq. (4.5) with $h = 0$, we observe that there

is a single meta-block μ_i at height i of T_i, and this meta-block is a Type M meta-block (cf. the right side of Fig. 4.16). Our claim is that prepending $\mathtt{a}$ to Y changes the first and the last block of every μ_i $(0 \le i \le k-1)$: The prepended $\mathtt{a}$ forms a Type 2 meta-block with the first character of X_0 by "stealing" the first character from μ_0, and this character is a $\beta_0 = \mathtt{b}$. Assume that μ_i $(0 \le i \le k-1)$ loses its first symbol (i.e., β_i). By relinquishing this symbol, μ_i becomes a Type 1 meta-block consisting only of α_i's. The last two α_i's contained in μ_i are grouped into a block α'_{i+1} of length *two*, where $\alpha'_{i+1} \coloneqq \langle \alpha_i \alpha_i \rangle_1[1]$ is the name of the root node of $\mathsf{ET}(\alpha_i \alpha_i)$. Every newly appearing node α'_{i+1} gets combined with its right-adjacent node β_{i+1} to form a new Type 2 meta-block. The used β_{i+1} is stolen from μ_{i+1}, and hence we observe an iterative process of stealing the first symbol β_{i+1} from μ_{i+1} for each height $i = 0, \ldots, k-2$. Figure 4.44 visualizes this observation on the lowest two heights.

This observation can be inductively proven for each even integer i with $0 \le i \le h-2$. By Eq. (4.5), we know that $\langle X_i \rangle_i = \beta_i \alpha_i^{3^{k-i}-1}$ and $\langle X_{i+1} \rangle_i = \beta_i^3 \alpha_i^{3^{k-i}-3}$. Then

$$\begin{aligned}
\mathsf{esp}(\mathsf{esp}(\langle X_i \rangle_i \langle X_{i+1} \rangle_i)) &= \mathsf{esp}\Big(\mathsf{esp}\Big(\beta_i \alpha_i^{3^{k-i}-1} \beta_i^3 \alpha_i^{3^{k-i}-3}\Big)\Big) \\
&\overset{\text{(Cor. 4.12)}}{=} \mathsf{esp}\Big(\mathsf{esp}(\beta_i \alpha_i \alpha_i)\, \mathsf{esp}\Big(\alpha_i^{3^{k-i}-3}\Big)\, \beta_{i+1} \alpha_{i+1}^{3^{k-i-1}-1}\Big) \\
&= \mathsf{esp}\Big(\mathsf{esp}(\beta_i \alpha_i \alpha_i)\, \alpha_{i+1}^{3^{k-i-1}-1} \beta_{i+1} \alpha_{i+1}^{3^{k-i-1}-1}\Big) \\
&\overset{\text{(Cor. 4.12)}}{=} \mathsf{esp}\Big(\mathsf{esp}(\beta_i \alpha_i \alpha_i)\, \alpha_{i+1}^{3^{k-i-1}-1}\Big)\, \mathsf{esp}\Big(\beta_{i+1} \alpha_{i+1}^{3^{k-i-1}-1}\Big) \\
&= \mathsf{esp}\Big(\mathsf{esp}(\beta_i \alpha_i \alpha_i)\, \alpha_{i+1}^{3^{k-i-1}-1}\Big)\, \mathsf{esp}(\beta_{i+1} \alpha_{i+1} \alpha_{i+1}) \\
&\quad\ \mathsf{esp}\Big(\alpha_{i+1}^{3^{k-i-1}-3}\Big),
\end{aligned}$$

and $\mathsf{esp}(\alpha_{i+1}^{3^{k-i-1}-3}) = \alpha_{i+2}^{3^{k-i-2}-1}$. Prepending α'_i (set $\alpha'_0 \coloneqq \mathtt{a}$) to the string $\langle X_i \rangle_i \langle X_{i+1} \rangle_i$ yields

$$\begin{aligned}
\mathsf{esp}(\mathsf{esp}(\alpha'_i \langle X_i \rangle_i \langle X_{i+1} \rangle_i)) &= \mathsf{esp}\Big(\mathsf{esp}\Big(\alpha'_i \beta_i \alpha_i^{3^{k-i}-1} \beta_i^3 \alpha_i^{3^{k-i}-3}\Big)\Big) \\
&\overset{\text{(Cor. 4.12)}}{=} \mathsf{esp}\Big(\mathsf{esp}(\alpha'_i \beta_i)\, \mathsf{esp}\Big(\alpha_i^{3^{k-i}-1}\Big)\, \beta_{i+1} \alpha_{i+1}^{3^{k-i-1}-1}\Big) \\
&= \mathsf{esp}\Big(\mathsf{esp}(\alpha'_i \beta_i)\, \mathsf{esp}\Big(\alpha_i^{3^{k-i}-3} \alpha_i \alpha_i\Big)\, \beta_{i+1} \alpha_{i+1}^{3^{k-i-1}-1}\Big) \\
&= \mathsf{esp}\Big(\mathsf{esp}(\alpha'_i \beta_i)\, \alpha_{i+1}^{3^{k-i-1}-1} \alpha'_{i+1} \beta_{i+1} \alpha_{i+1}^{3^{k-i-1}-1}\Big) \\
&\overset{\text{(Cor. 4.12)}}{=} \mathsf{esp}\Big(\mathsf{esp}(\alpha'_i \beta_i)\, \alpha_{i+1}^{3^{k-i-1}-1}\Big)\, \mathsf{esp}\Big(\alpha'_{i+1} \beta_{i+1}\Big) \\
&\quad\ \mathsf{esp}\Big(\alpha_{i+1}^{3^{k-i-1}-3} \alpha_{i+1} \alpha_{i+1}\Big) \\
&= \mathsf{esp}\Big(\mathsf{esp}(\alpha'_i \beta_i)\, \alpha_{i+1}^{3^{k-i-1}-1}\Big)\, \mathsf{esp}\Big(\alpha'_{i+1} \beta_{i+1}\Big) \\
&\quad\ \alpha_{i+2}^{3^{k-i-2}-1} \alpha'_{i+2},
\end{aligned}$$

and α'_{i+2} carries on to the nodes $\langle X_{i+2} \rangle_{i+2} \langle X_{i+3} \rangle_{i+2}$ on height $i+2$ due to Cor. 4.12.

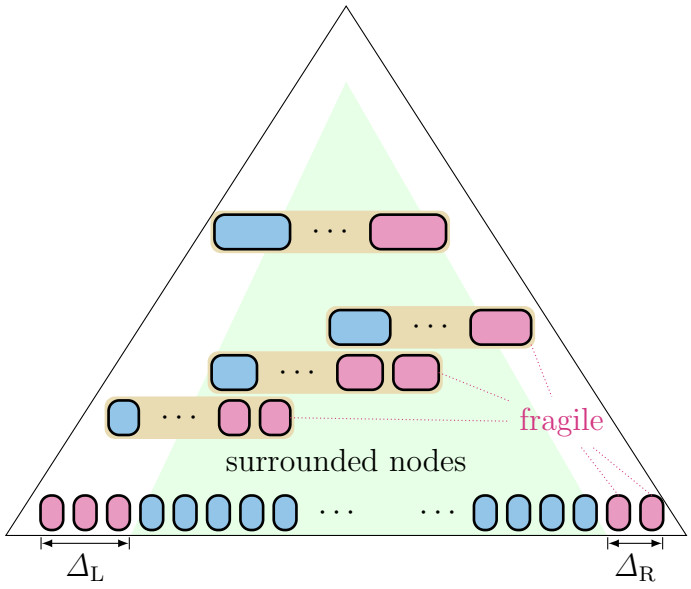


Fig. 4.17: Division of an ESP tree in surrounded and fragile nodes. The surrounded nodes form an inner cone. Neighboring fragile nodes can appear in the non-surrounded areas (e.g., the lowest leftmost nodes). On each height, the ESP tree can have a constant number of fragile surrounded nodes that do not have fragile nodes in their subtrees.

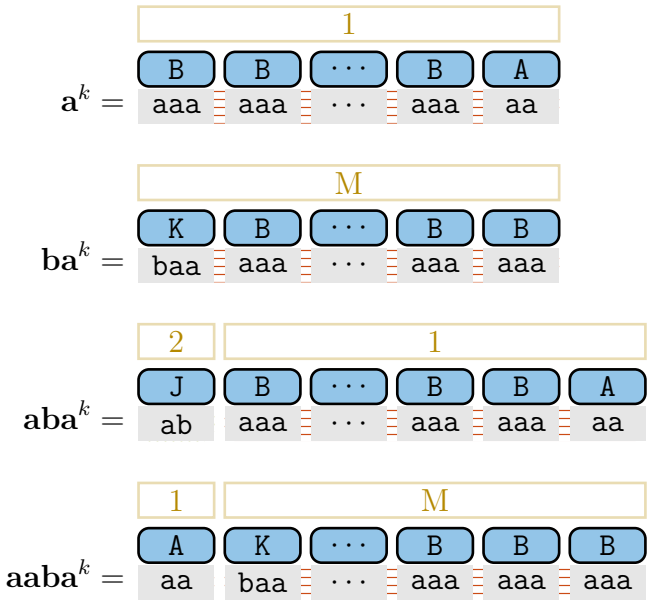


Fig. 4.18: Prepending the string aab to the text $\mathtt{a}^k$ character by character. Each step is given as a row, in which we additionally computed the ESP of the current text. The last row shows an example, where a former Type 1 meta-block changes to Type M, although it is right of a Type 2 meta-block. Here, $k \mod 3 = 2$.

Overall, the leftmost and rightmost node on height $i+1$ of T_i changes, for $i = 0, \ldots, k-1$. Such a changed node v of T_i on height $i+1$ has $k-i-1$ ancestors in T_i, which become changed by changing the name of v. Therefore, at least $k-1+(k-2)+(k-3)+\cdots+1 = (k^2-k)/2$ nodes are changed. Hence there is a lower bound of $\Omega(k^2) = \Omega(\log_3^2(n/\log_3 n)) = \Omega(\lg^2 n)$ fragile nodes. □

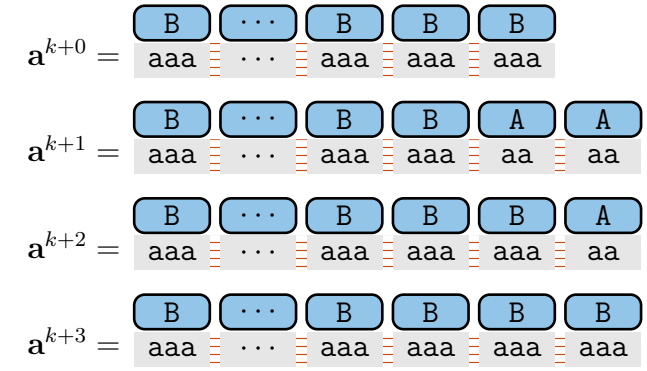
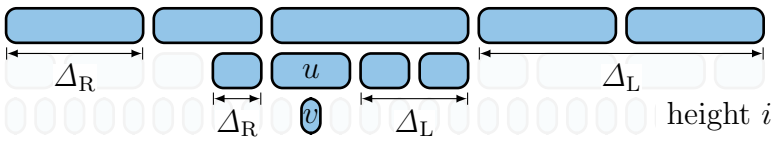


Fig. 4.19: Greedy blocking of a Type 1 meta-block. The greedy blocking is related to the Euclidean division by three. The remainder $k \bmod 3$ is determined by the number of symbols in the last two blocks (here, $k \bmod 3 = 0$). In this example, the ESP technique creates a single, repeating meta-block on each input.

A new upper bound. With Thms. 4.11 and 4.13, we conclude that the $\mathcal{O}(\lg^* n \lg n)$-bound on the number of fragile nodes for square-free strings (Lemma 4.8) does not hold for general strings. To obtain a general upper bound (we stick again to Rule (M)), we include the repeating meta-blocks in our study of fragile nodes. Fragile nodes can now be surrounded (trees of square-free strings do not have fragile surrounded nodes according to Lemma 4.8). Remembering that a node is fragile if it has a fragile child, a fragile Type 2 node can also be surrounded (e.g., one of its children can be a fragile surrounded repeating node). Figure 4.17 sketches the possible occurrences of fragile surrounded nodes. A first result on a special case is given in the following lemma:

Lemma 4.14. A surrounded node v is contained in the local surroundings of $\mathcal{O}(\lg^* n \lg n)$ nodes. If all those nodes are of Type 2, then a change of v causes $\mathcal{O}(\lg^* n \lg n)$ name changes.

Proof. We follow [55, Proof of Lemma 9]: We count the number of nodes that contain v in its local surrounding. Given that v is a node on height i and u is v's parent, then there are at most $\Delta_R/2 \leq \Delta_R$ nodes preceding u and $\Delta_L/2 \leq \Delta_L$ nodes succeeding u that have v in its local surrounding. We count one on height i, and $(\Delta_L + \Delta_R + 1)/2$ on height $i+1$. Since the counted nodes on height $i+1$ are consecutive, there are at most $(\Delta_L + \Delta_R + 1)/2$ nodes that are all parents of the counted nodes on height $i+1$. Consequently, there are at most $(\Delta_L + \Delta_R + 1)/2 + \Delta_L + \Delta_R$ nodes on height $i+2$ that have v in their local surroundings. Iterating over all heights gives an upper bound of $(\Delta_L + \Delta_R + 1) \sum_{h=0}^{\lg n - i} 1/2^h \leq 2(\Delta_L + \Delta_R + 1)$ nodes on each height. □



Second, we narrow down the fragile blocks in repeating meta-blocks. The first block (cf. Fig. 4.18) and the two rightmost blocks (cf. Fig. 4.19) of a repeating meta-block can be fragile. Due to the greedy parsing, all other blocks

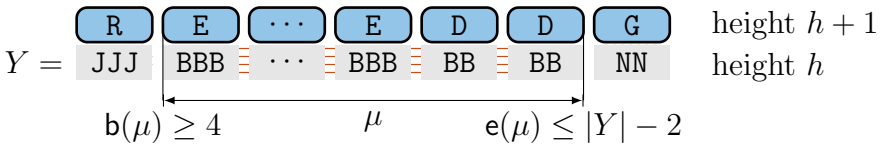


Fig. 4.20: Setting of Lemma 4.15. According to Lemma 4.15, a meta-block μ in $\mathsf{esp}(Y)$ of a string Y cannot contain a surrounded fragile block if $\mathsf{b}(\mu) \geq 4$ and $\mathsf{e}(\mu) \leq |Y| - 2$.

of a repeating meta-block are (semi-)stable. A repeating meta-block containing fragile *surrounded* blocks needs to cover one of the leftmost or rightmost symbols, as can be seen by the following lemma:

Lemma 4.15. A repeating meta-block μ of $\mathsf{esp}(Y)$ with $\mathsf{b}(\mu) \geq 4$ and $\mathsf{e}(\mu) \leq |Y| - 2$ cannot contain a fragile block.

Proof. Since $\mathsf{b}(\mu) \geq 4$, there are at least three symbols before μ that are assigned to one or more other meta-blocks. When prepending symbols, those meta-blocks can change, absorbing the new symbols or giving the leftmost symbol away to form a Type 2 meta-block. In neither case, they can affect the parsing of μ, since μ is parsed greedily. Similarly, the succeeding meta-blocks of μ keep μ's blocks from changing when appending symbols. See Fig. 4.20 for a sketch. □

Corollary 4.16. The edit sensitive parsing introduces at most two fragile surrounded blocks. These blocks are the two rightmost blocks of a repeating meta-block whose leftmost block is not surrounded.

Lemma 4.17. Changing the symbol in a substring of $\langle Y \rangle_{h-1}$ on which a repeating node on height h is built changes $\mathcal{O}(1)$ names on height h.

Proof. Let u be a repeating node on height h. Since it is repeating, it is built on a substring $X \coloneqq \langle Y \rangle_{h-1}[\mathsf{b}(X) \,..\, \mathsf{e}(X)]$ of a repeating meta-block $\mu = \langle Y \rangle_{h-1}[\mathsf{b}(\mu) \,..\, \mathsf{e}(\mu)]$ with $\mathfrak{D}(u) = X$. Now change a symbol in X, say $\langle Y \rangle_{h-1}[i_u]$ with $\mathsf{b}(X) \leq i_u \leq \mathsf{e}(X)$. This causes the name of u to change. Additionally, it causes the meta-block μ to split into a repeating meta-block $\langle Y \rangle_{h-1}[\mathsf{b}(\mu) \,..\, i_u - 1]$ and a Type M meta-block $\langle Y \rangle_{h-1}[i_u \,..\, \mathsf{e}(\mu)]$, causing the names of the two rightmost nodes built on the new meta-blocks to change. Altogether, there are $\mathcal{O}(1)$ name changes on height h. □

An easy generalization of Lemma 4.17 is that changing k consecutive nodes on height $h-1$ that are children of repeating nodes on height h changes $\mathcal{O}(k)$ names on height h. With Lemma 4.17, the following lemma translates the result of Cor. 4.16 for blocks to nodes:

Lemma 4.18. The ESP tree $\mathsf{ET}(Y)$ of a string Y of length n has $\mathcal{O}(\lg^2 n \lg^* n)$ fragile nodes, and $\mathcal{O}(h \lg^* n)$ fragile nodes on height h.

Proof. While computing $\langle Y \rangle_{h+1}$ from $\langle Y \rangle_h$, the ESP technique introduces $\mathcal{O}(1)$ fragile surrounded blocks according to Cor. 4.16. Each fragile surrounded block corresponds to a fragile surrounded node.

Similar to the proof of Lemma 4.8, we count all surrounded nodes as fragile whose local surrounding contains a fragile node. Lemma 4.14 shows that each introduced fragile surrounded block makes $\mathcal{O}(\lg^* n \lg n)$ nodes fragile. Although we considered only Type 2 nodes in Lemma 4.14, we can generalize this result for all fragile nodes with Lemma 4.17.

To sum up, there are $\mathcal{O}(h \lg^* n)$ fragile nodes on height h. Because $\mathsf{ET}(X)$ has a height of at most $\lg n$, there are $\mathcal{O}(\lg^* n \sum_{h=1}^{\lg n} h) = \mathcal{O}(\lg^* n \lg^2 n)$ fragile nodes in total. □

Showing that the number of fragile nodes is indeed larger than assumed makes ESP trees a more unfavorable data structure, since fragile nodes are cumbersome when comparing strings with ESP trees as done in [55]. Fortunately, we can restore the claimed number of $\mathcal{O}(\lg n \lg^* n)$ fragile nodes for a string of length n with a slight modification of the parsing, as shown in the following section.

4.3 Hierarchical Stable Parsing Trees

Our modification, which we call *hierarchical stable parsing (HSP)*, augments each name with a *surname* and a *surname-length*, whose definitions follow: Given a name $\mathtt{Z} \in \Sigma_h$, let h' with $0 \leq h' \leq h$ be the largest integer such that $\mathfrak{D}^{(h')}(\mathtt{Z})$ consists of the same symbol, say $\mathfrak{D}^{(h')}(\mathtt{Z}) = \mathtt{Y}^\ell$ for a symbol $\mathtt{Y} \in \Sigma_{h-h'}$ and an integer $\ell \geq 1$. Then the surname and surname-length of $\mathtt{Z}$ are the symbol $\mathtt{Y}$ and the integer ℓ, respectively.[2] For convenience, we define the surname of a character to be the character itself. Then all symbols in $\mathfrak{D}^{(j)}(\mathtt{Z})$ for every j with $1 \leq j \leq h'$ share the same surname with $\mathtt{Z}$.

Having the surnames of the nodes at hand, we present the HSP. It differs from ESP in how a string of names is partitioned into meta-blocks, whose boundaries now depend on the surnames: When factorizing a string of names into meta-blocks, we relax the check whether two names are equal; instead of comparing names we compare by surnames.[3] As a consequence, we allow meta-blocks of Type 1 to contain different symbols as long as all symbols share the same surname. The other parts of the edit sensitive parsing defined in Sect. 4.2.2 are left untouched; in particular, the alphabet reduction uses the symbols as before. We write $\mathsf{HT}(Y)$ for the resulting parse tree, called *HSP tree*, when the HSP technique is applied to a string Y. Figure 4.21 shows $\mathsf{HT}(\mathtt{a}^{11}(\mathtt{ba})^5)$. In the rest of this chapter (and as shown in Fig. 4.21), we give a repetitive node with surname $\mathtt{Z}$ and surname-length ℓ the name $\mathtt{Z}_\ell$. We omit the surname-length if it is one (and thus, the label of a non-repetitive node is equal to its name).

[2] By definition, the surname of $\mathtt{Z}$ is $\mathtt{Z}$ itself if $\ell = 1$.

[3] The check is relaxed since names with different surnames cannot have the same name.

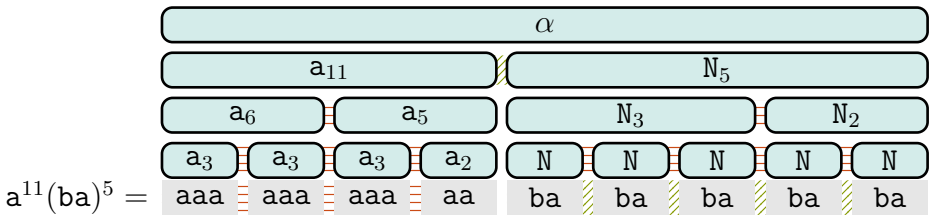


Fig. 4.21: Hierarchical stable parsing. The repeating meta-blocks are determined by the surnames.

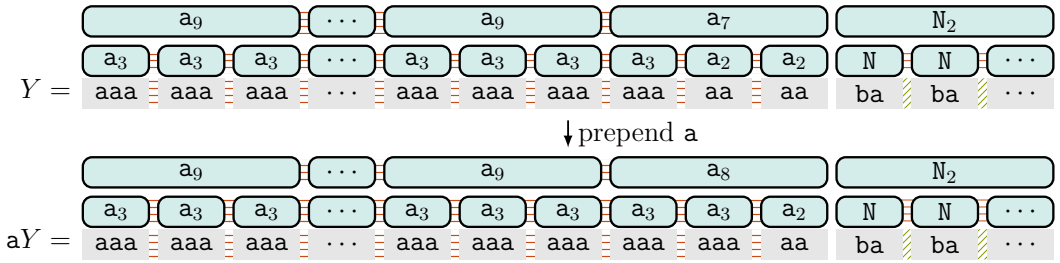


Fig. 4.22: Excerpt of $\mathsf{HT}(Y)$ (*upper part*) and $\mathsf{HT}(\mathtt{a}Y)$ (*lower part*), where $Y = \mathtt{a}^k(\mathtt{ba})^{k'}$ with $k = 18 + 9^i + 7$ for an integer $i \geq 0$ and $k' \geq 2$ (cf. Fig. 4.10). The parsing of Y creates a repeating meta-block consisting of $\mathtt{a}^k$, and a Type 2 meta-block consisting of $(\mathtt{ba})^2$. For $k \geq 2$ it is impossible to modify the latter meta-block by prepending characters, since the parsing always groups adjacent nodes with the same surname into one repeating meta-block.

For the other nodes, we use the names of Fig. 4.6. We can do that because the name of a node can be identified by its surname and surname-length, as can be seen by the following lemma:

Lemma 4.19. The name of a node is uniquely determined by its surname and surname-length.

Proof. A node with surname-length one is not repetitive, and therefore, its name is equal to its surname. Given a repetitive node v with surname $\mathtt{Z}$ and surname-length ℓ, there is a height h such that $\mathfrak{D}^{(h)}(v) = \mathtt{Z}^\ell$. For every height h' with $1 \leq h' \leq h$, $\mathfrak{D}^{(h')}(v)$ consists of the same symbol, and hence $\mathfrak{D}^{(h')}(v)$ is parsed greedily by HSP. Consequently, the iterated greedy parsing of the string $\mathtt{Z}^\ell$ determines the name of v. □

4.3.1 Upper Bound on the Number of Fragile Nodes

The motivation of introducing the HSP technique becomes apparent with the three following facts:

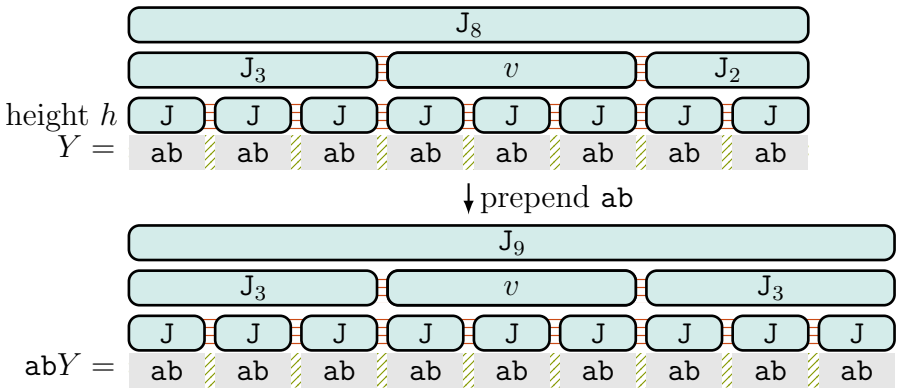


Fig. 4.23: Comparison of $\mathsf{HT}(Y)$ and $\mathsf{HT}(\mathtt{ab}Y)$, where $Y = (\mathtt{ab})^8$. The node v with name $\mathtt{J}_3$ is semi-stable.

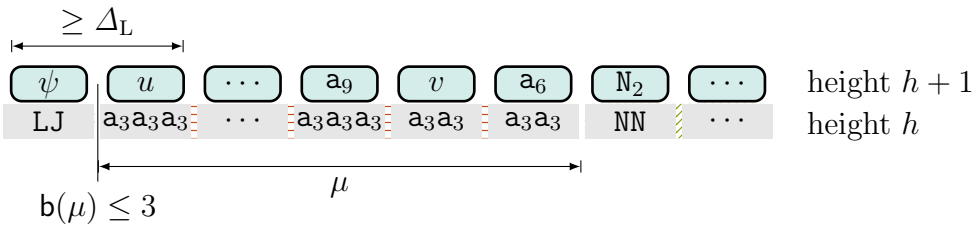


Fig. 4.24: Setting of Cor. 4.20. According to Lemma 4.15, a meta-block μ can contain a surrounded fragile block if $\mathsf{b}(\mu) \leq 3$ (cf. Fig. 4.20). In the figure, the node v is fragile, since prepending $\mathtt{L}$ changes its name. According to Cor. 4.20, there is a non-surrounded node u whose generated substring has the generated substring of v as a prefix.

Fact 1: Given that the surnames of the repetitive nodes in a repeating meta-block μ are w, the generated substring of each such repetitive node is a repetition of the form X^k with the same $X = \mathsf{string}(w) \in \Sigma^*$ (or $X = w$ in case $w \in \Sigma$), but with possibly different surname-lengths k (e.g., $\mathsf{string}(\mathtt{N}_3) = (\mathtt{ba})^3$ and $\mathsf{string}(\mathtt{N}_2) = (\mathtt{ba})^2$ in Fig. 4.21). Due to the greedy parsing of the repeating meta-blocks, the surname-lengths of the last two nodes in μ cannot be larger than the surname-lengths of the generated substrings of the other nodes (with the same surname) contained in μ. See Fig. 4.22 for an example when prepending a character to the input (observe that $\mathtt{a}_7$ changes to $\mathtt{a}_8$, whose generated substring is still a prefix of $\mathsf{string}(\mathtt{a}_9)$).

Fact 2: The shift of a semi-stable node is always a multiple of the length of its surname (recall that semi-stable nodes are defined like stable nodes, but with slight shifts, cf. Sect. 4.2.4): Let $\mathtt{J}$ be the surname of a semi-stable node $v \in \langle Y \rangle_h$ on height h. Given $\mathtt{J} \in \Sigma_{h'}$ for a height h' with $h' \geq 0$, $\mathfrak{D}^{(h-h')}(v)$ is a repetition of the symbol $\mathtt{J}$ on height h'. A shift of v can

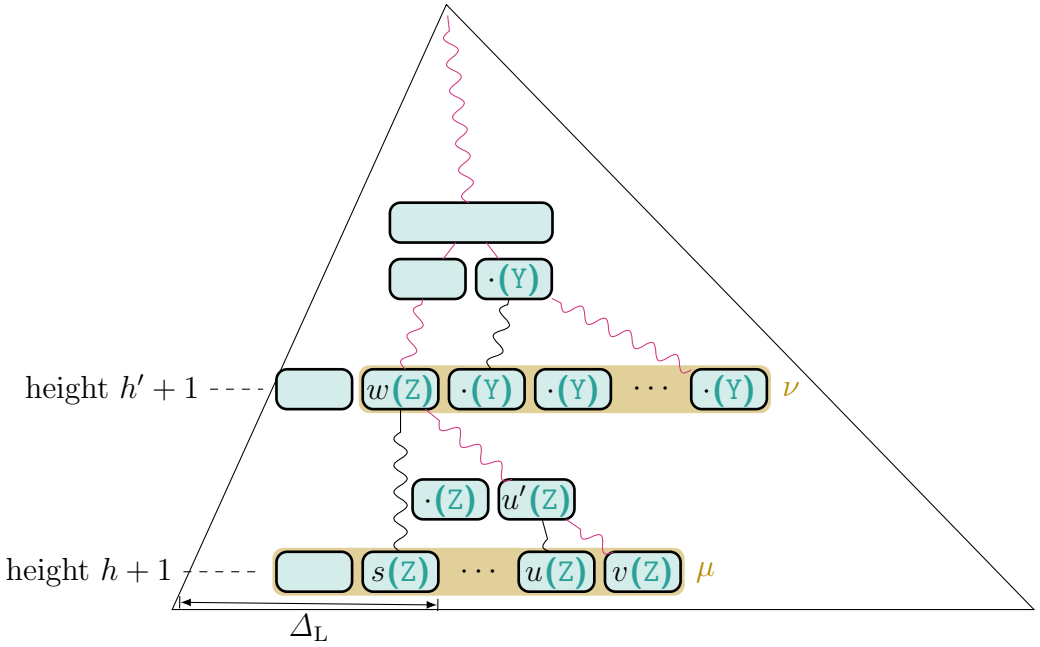


Fig. 4.25: Sketch of the HSP tree used to show Lemma 4.22. In the sketch, we give the repetitive nodes of the meta-block ν the surname Y. Repetitive nodes are labeled with their surnames, which are put into parentheses.

only be caused by adding one or more Js to $\langle Y \rangle_{h'}$. In other words, the shift is always a multiple of $\mathfrak{D}^{(h')}(\mathtt{J})$. Figure 4.23 shows an example of a semi-stable node v.

Fact 3: A non-repetitive Type M block can be fragile only if it is non-surrounded. By definition, a repeating meta-block μ contains a non-repetitive block β if and only if μ is Type M. The block β can only be located at the beginning or ending of μ. Remembering Rule (M), β's non-repetitiveness is caused by

- fusing a symbol with its *succeeding* meta-block, or
- fusing the *last* symbol with its *preceding* meta-block.

In both cases, it is impossible that β is a surrounded block if $\mathsf{b}(\mu) \leq \Delta_{\mathrm{L}}$. If β is surrounded, it is (semi-)stable due to Lemma 4.15. With Rule (M), we also experience a more stable behavior like in Fig. 4.45.

These facts make the HSP technique more stable than the ESP technique, as can be seen in Fig. 4.46, for instance. In the following, we study the number of fragile surrounded nodes (like in Sect. 4.2.4 for the ESP trees), and show the invariant (Claim 3 in Lemma 4.22) that the generated substring of a fragile surrounded node is always the prefix of the generated substring of a name that is already stored in $\mathfrak{D}$. On block level, this is an easy conclusion of Lemma 4.15 and Facts 1 and 3.

Corollary 4.20. Given $n > 4$ and a repeating meta-block μ having a fragile surrounded block β, μ has at least one block preceding β that contains three symbols with the same surname. In particular, the leftmost of these preceding blocks is non-surrounded.

Proof. Since β is surrounded and fragile, $\mathsf{b}(\mu) \leq 2$ according to Lemma 4.15 and Cor. 4.16. Hence, $|\mu| \geq \Delta_{\mathrm{L}} - 2$ (otherwise β would not be surrounded). By the definition of Δ_{L} in Lemma 4.5, $\Delta_{\mathrm{L}} - 2 \geq 5$ for $n > 4$. Assuming that the repetitive blocks in μ have the surname $\mathtt{Z}$, there is at least one repetitive block γ with surname $\mathtt{Z}$ preceding β that contains three symbols of μ. But the fragile surrounded block β is also a repetitive block according to Fact 3. Due to Fact 1, the surname-length of β is at most as long as the surname-length of γ, i.e., the generated substring of the node corresponding to β is a prefix of the generated substring of the node corresponding to γ. Let γ be the leftmost such block. Remembering that μ can start with a non-repetitive node in case that μ is of Type M, it is not obvious that γ is non-surrounded. However, we know that $\mathsf{b}(\mu) \leq 2$. Hence, $\mathsf{b}(\gamma) \leq 5 \leq \Delta_{\mathrm{L}}$, yielding that γ is non-surrounded. See Fig. 4.24 for a sketch (with $\mathtt{Z} = \mathtt{a}$). □

In general, the aforementioned invariant does not hold for ESP trees, but is essential for the sparse suffix sorting in text space. There, our idea is to create an HSP or ESP tree on a newly found re-occurring substring. We would like to store the ESP tree in the space of one of those substrings, which we can do by truncating the tree at a certain height (removing the lower nodes), and changing the pointer of each (new) leaf such that the name of a leaf refers to its generated substring that is found in the remaining text. Unfortunately, there is a problem when pre-/appending characters to enlarge the ESP tree, since a leaf could change its name such that its generated substring needs to be updated - which can be non-trivial if its generated substring refers to an already overwritten part of the text that is not present in the remaining text as a (complete) substring. Figure 4.47 demonstrates the problem when truncating ESP trees at height 2. Fortunately, the following lemmas restrict the problem of updating the generated substring when an HSP node is surrounded and fragile. We start with appending characters:

Lemma 4.21. There is no surrounded HSP node v whose name changes when *appending* characters.

Proof. Assume that v's name changes on appending characters. Moreover, assume that v's local surrounding does not contain a fragile node (otherwise swap v with this node). First, since there is no fragile node in v's local surrounding, it has to be a repeating node according to Lemma 4.7. Second, according to Cor. 4.16, it has to be one of the last two nodes built on a repeating meta-block μ. But there is no way to change the names of the last two blocks of μ by appending characters unless these blocks are non-surrounded. So a

surrounded node cannot have a node in its surrounding whose name changes when appending characters. □

Lemma 4.22. Let v be a fragile surrounded node of an HSP tree. Then

Claim 1: v is a repetitive node,

Claim 2: pre-/appending characters cannot change v's surname, and

Claim 3: the generated substring of v is always a prefix of the generated substring of an already existing node belonging to the same meta-block as v.

Proof. To show the lemma, let $n > \Delta_\mathrm{L} + \Delta_\mathrm{R}$, otherwise there are no surrounded nodes. There are two (non-exclusive) possibilities for a node to be fragile and surrounded:

- it belongs to the last two nodes built on a repeating meta-block (due to Cor. 4.16), or
- its subtree contains a fragile surrounded node, since by definition,
  - a node is fragile if it contains a fragile node in its subtree, and
  - all nodes in the subtree of a surrounded node are surrounded.

We iteratively show the claim for all heights, starting at the bottom: Let v be one of the *lowest* fragile surrounded nodes in $\mathsf{HT}(Y)$ (*lowest* meaning that there is no fragile node in v's subtree). Suppose that v is a node on height $h+1$ with $h \geq 0$. Since there is no fragile surrounded node in v's subtree, v is one of the last two nodes built on a repeating meta-block $\langle Y \rangle_h[\mu]$ (i.e., $Y[\mu]$ for $h = 0$). Due to Fact 3, Claim 1 holds for v; let $\mathtt{Z}$ be its surname. Since v is fragile, $\mathsf{b}(\mu) \leq 3$ must hold (otherwise we get a contradiction to Lemma 4.15). But since v is surrounded, there is a repetitive node u with surname $\mathtt{Z}$ preceding v that is built on three symbols ($\mathfrak{D}(u) \in \Sigma_h^3$) of μ due to Cor. 4.20. In particular, the leftmost repetitive node s of μ is not surrounded.

We only consider prepending a character (appending is already considered in Lemma 4.21). Assume that v's name changes when prepending a specific character. By Fact 1, the HSP technique assigns a new name to v, but it does not change its surname (so Claim 2 holds for v). Additionally, $\mathsf{string}(v)$ is a substring of $\mathsf{string}(u)$, where u is one of v's preceding nodes having the surname $\mathtt{Z}$, and therefore Claim 3 holds for v. For example, let v be the node with name $\mathtt{a}_7$ in $\mathsf{HT}(Y)$ of Fig. 4.22, then $\mathsf{string}(v) = \mathtt{a}^7$, which is a prefix of $\mathsf{string}(\mathtt{a}_9) = \mathtt{a}^9$. After prepending the character $\mathtt{a}$, v's name becomes $\mathtt{a}_8$ with $\mathsf{string}(v) = \mathtt{a}^8$. Still, $\mathsf{string}(v)$ is a prefix of $\mathsf{string}(\mathtt{a}_9)$.

Due to this behavior, the node v is always assigned to μ, regardless of what character is prepended. It is only possible to extend or shorten μ on its left side, or equivalently, μ's right end is *fixed*; the parsing of a meta-block succeeding μ

cannot change. Put differently, the parsing assures that *every surrounded* node located to the right of $\langle Y \rangle_h[\mu]$ is (semi-)stable. We conclude that the claim holds for the heights $1, \ldots, h+1$.

Next, we show that the claim holds for all height $h+2, \ldots, h'$, where $h'+1$ is the height of the LCA w of s and v. Figure 4.25 gives a visual representation of the following observations: When following the nodes from v up to w, there is a path of ancestor nodes with surname Z. Except for w, each such ancestor node u' has a neighbor with surname Z. On changing the name of v, all nodes on the height of u' are unaffected, except u'. That is because the ancestor of s on the same height as u' is put with u' in the same repeating meta-block, which comprises all neighboring nodes with surname Z. By the analysis above, changing the name of u' cannot change the parsing of the other nodes on the same height. We conclude that the claim holds for the heights $h+2, \ldots, h'$.

Let us focus on the nodes on height $h'+1$: The node w is not surrounded, because it contains the non-surrounded node s in its subtree. Having neighbors with different surnames, w is either blocked in a Type 2 or Type M meta-block.

- In the former case (Type 2), the analysis of Lemma 4.14 shows that w only affects the parsing of the non-surrounded nodes. There can be a non-surrounded meta-block on a height $h'' > h'+1$ having a fragile surrounded node v'. But then v' cannot contain a fragile node (the descendants of w are the last fragile *surrounded* nodes, and w is non-surrounded). Hence, we can apply the same analysis to v' as for v.

- In the latter case (Type M), w is fused with a repeating meta-block to form a Type M meta-block ν, changing the names of the leftmost and two rightmost nodes of ν, where the leftmost node is w. Assume that the two rightmost nodes of ν are fragile and surrounded (otherwise we conclude with the previous case that there are no fragile surrounded nodes on height $h'+1$). Under this assumption, the rightmost nodes of ν are repeating nodes due to Fact 3. Hence, we can apply the same analysis as for v, and conclude the claim for all heights above h'. □

A direct consequence is that there are $\mathcal{O}(1)$ fragile surrounded nodes on each height. We can adapt the result of Lemma 4.18 to HSP trees, and obtain the following theorem:

Theorem 4.23. The HSP tree $\mathsf{HT}(Y)$ of a string Y of length n contains at most $\mathcal{O}(\lg^* n)$ fragile nodes on each height.

Having a bound on the number of fragile nodes, we start to study the algorithmic operations of an HSP tree. The first operation is how to actually build an HSP tree. For that, we have to think about its representation:

4.3.2 Tree Representation

Unlike Cormode and Muthukrishnan, who use hash tables to represent the dictionary $\mathfrak{D}$, we follow a deterministic approach. In our approach, we represent $\mathfrak{D}$ by storing the HSP tree as a CFG. A name (i.e., a non-terminal of the CFG) is represented by a pointer to a data field (an allocated memory area), which is composed differently for leaves and internal nodes:

Leaves. A leaf stores a position i and a length $\ell \in \{2, 3\}$ such that $Y[i..i+\ell-1]$ is the generated substring.

Internal nodes. An internal node stores the length of its generated substring, and the names of its children. If it has only two children, we use a special, invalid name $\perp$ for the non-existing third child such that all data fields are of the same length.

This information helps us to navigate from a node to its children or its generated substring in constant time, and to navigate top-down in the HSP tree by traversing the tree from the root in time linear in the height of the tree.

To accelerate substring comparisons, we want to give nodes with the same children (with respect to their order and names) the same name, such that the dictionary $\mathfrak{D}$ is injective. To keep the dictionary injective, we do the following: Before creating a new name for the rule $b \to xyz$ (we set $z = \perp$ if the rule is $b \to xy$), we check whether there already exists a name for xyz. To perform this lookup efficiently, we need also the *reverse* dictionary of $\mathfrak{D}$, with the right hand side of the rules as search keys. We want the reverse dictionary to be of size $\mathcal{O}(|Y|)$, supporting lookup and insert in $\mathcal{O}(t_{\text{look}})$ (deterministic) time for a $t_{\text{look}} = t_{\text{look}}(n)$ depending on n. For instance, a balanced binary search tree has $t_{\text{look}} = \mathcal{O}(\lg n)$.

With this tree representation, we can build HSP trees within the following time and space bounds:

Lemma 4.24. The HSP tree $\mathsf{HT}(Y)$ of a string Y of length n can be built in $\mathcal{O}(n(\lg^* n + t_{\text{look}}))$ time. It takes $\mathcal{O}(n)$ words of space.

Proof. A name is inserted or looked-up in t_{look} time. Due to the alphabet reduction technique (see Lemma 4.4), applying esp on a substring of length ℓ takes $\mathcal{O}(\ell \lg^* n)$ time, returning a sequence of blocks of length at most $\ell/2$. □

4.3.3 LCE Queries with HSP Trees

The idea of devising LCE data structures based on the alphabet reduction is not new. Alstrup et al. [5, Thm. 2] considered building signature encoding parse trees on a set of strings such that the LCP of two strings of this set can be computed efficiently. Nishimoto et al. [198, Lemma 10] enhanced these parse trees with an algorithm computing LCE queries. Similar to these two

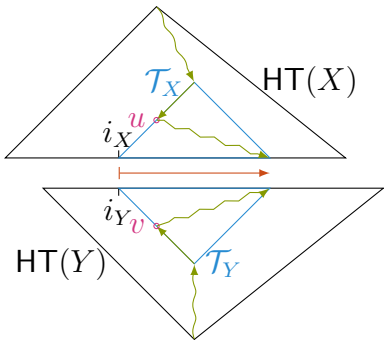


Fig. 4.26: Conception of the proof of Lemma 4.25. To compute the LCP of $X[i_X\,..]$ and $Y[i_Y\,..]$ (arrow in the center), we walk down the trees $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ (depicted by the upper and the lower triangle, respectively) on the paths towards the leaves containing $X[i_X]$ and $Y[i_Y]$, respectively, by simultaneously visiting two nodes on the same height of both trees. In this figure, each of these paths is depicted by a sequence of green arrows. The nodes u and v are on these paths. Suppose that they are on the same height and have the same surname. On visiting both nodes, we know that the LCP is at least $\min(|\mathsf{string}(u)|\,, |\mathsf{string}(v)|)$ long. We update the destination of our traversal accordingly, such that we follow the paths from u and v to the leaves covering the not-yet checked parts of the LCP that we want to compute.

approaches, we show that HSP trees are also good at answering LCE queries. The common idea of all LCE algorithms is to compare the names of two nodes to test whether the generated substrings of both nodes are the same. Remembering that two nodes with the same generated substring can have different names (cf. Sect. 4.2.3.1 and Fig. 4.9), we want to have a rule at hand saying when two nodes with different names must have different generated substrings. It is easy to provide such a rule when the input string is square-free: In this case, all fragile nodes are non-surrounded according to Lemma 4.8, and thus we know that the surrounded nodes are stable. Since each height consists of exactly one Type 2 meta-block, the equality of two substrings X and Y can be checked by comparing the names of two surrounded nodes whose generated substrings are X and Y, respectively. For general strings, we need to enhance this rule for repeating nodes. That is because the names of two repeating nodes at *the same height* already differ when the generated substring of one node is a proper prefix of the generated substring of the other node. Our idea (and here our approach differs from [5, 198]) is to compare two nodes not by their names but by their surnames and surname-lengths (we use the property described in Fact 2 of Sect. 4.3.1). With that idea we explain how HSP trees can answer LCE queries efficiently. For that, we assume that all HSP trees have a *common* dictionary $\mathfrak{D}$ that additionally stores the length of the string $\mathfrak{D}^{(h)}(\mathsf{Z})$ for each name $\mathsf{Z} \in \Sigma_h$,

Lemma 4.25. Given $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ built on two strings X and Y with

$|X| \leq |Y| \leq n$ and two text positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $\mathsf{lcp}(X[i_X\,..], Y[i_Y\,..])$ in $\mathcal{O}(\lg n \lg^* n)$ time.

Proof. We use the following property: If two nodes have the same surname $\mathtt{Z}$, then the generated substrings of both nodes are Z^i and Z^j, respectively, with the respective surname-lengths i and j, where $Z = \mathsf{string}(\mathtt{Z})$. In such a case, the generated substring of one node is a prefix of the generated substring of the other. In the particular case $i = j$, both nodes share the same subtree and consequently have the same name according to Lemma 4.19. In summary, this property allows us to omit the comparison of the subtrees of two nodes with the same surname, and thus speeds up the LCE computation, which is done in the following way (cf. Fig. 4.26):

(1) We start with traversing the two paths from the roots of $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ to the leaves λ_X and λ_Y whose generated substrings contain $\langle X\rangle_0[i_X]$ and $\langle Y\rangle_0[i_Y]$, respectively:

(2) We traverse the two paths leading to the leaves λ_X and λ_Y, respectively, in a simultaneous manner such that we always visit a pair (u, v) of nodes on the same height belonging to $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$, respectively.

(3) Given that u and v share the same surname $\mathtt{Z} \in \Sigma_h$, we know the lengths of their generated substrings $(\ell_u \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right|$ and $\ell_v \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right|)$ by having their surname-lengths ℓ_u and ℓ_v at hand. Given that i_u and i_v are the starting positions of $\mathsf{string}(u)$ and $\mathsf{string}(v)$, we know that $X[i_X\,..]$ and $Y[i_Y\,..]$ have a common prefix of at least

$$\min\Big(\ell_u \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right| - (i_u - i_X), \ell_v \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right| - (i_v - i_Y)\Big). \tag{4.6}$$

We update the variables λ_X and λ_Y to be the leaves whose generated substrings contain $\langle X\rangle_0\Big[i_u + \ell_u \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right|\Big]$ and $\langle Y\rangle_0\Big[i_v + \ell_v \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right|\Big]$, respectively.[4] Subsequently, we continue our tree traversals from u and v to the updated destinations λ_X and λ_Y, respectively. Since λ_X and λ_Y are no longer in the respective subtrees of u and v, we climb up the tree to the LCA of u (resp. v) and λ_X (resp. λ_Y), and recurse on (2).

(4) If we end up at a pair of leaves (i.e., $u = \lambda_X$ and $v = \lambda_Y$), we compare their generated substrings naïvely. If we find a mismatching character in both generated substrings, we can determine the value of ℓ and terminate. We also terminate if there is no mismatch, but λ_X or λ_Y is the rightmost leaf of $\mathsf{HT}(X)$ or $\mathsf{HT}(Y)$, respectively. In all other cases, we set λ_X and λ_Y to their respectively succeeding leaves, climb up to the parents of u and v, and recurse on (2).

[4] Instead of selecting the leaves whose generated substrings start at the end of the common prefix calculated in Eq. (4.6), we bookkeep the difference between $\ell_u \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right| - (i_u - i_X)$ and $\ell_v \left|\mathfrak{D}^{(h)}(\mathtt{Z})\right| - (i_v - i_Y)$.

During the traversals of both trees, we spend constant time for each navigational operation, i.e., (a) selecting a child, and (b) climbing up to the parent of a node: On the one hand, we select a child of a node v in constant time by following the pointer of the name of v (defined in Sect. 4.3.2). On the other hand, we maintain, for each tree, a stack storing all ancestors of the currently visited node during the traversal of the respective tree: Each stack uses $\mathcal{O}(\lg n)$ words, and can return the parent of the currently visited node in constant time.

To upper bound the running time of the traversals, we examine the nodes visited during the traversals. Starting at both root nodes, we follow the path from the root of $\mathsf{HT}(X)$ (resp. $\mathsf{HT}(Y)$) down to the roots of the minimal subtree $\mathcal{T}_X$ of $\mathsf{HT}(X)$ (resp. $\mathcal{T}_Y$ of $\mathsf{HT}(Y)$) covering $X[i_X \mathinner{..} i_X + \ell]$ (resp. $Y[i_Y \mathinner{..} i_Y + \ell]$).[5] After entering the subtrees $\mathcal{T}_X$ and $\mathcal{T}_Y$, we will never visit nodes outside of $\mathcal{T}_X$ and $\mathcal{T}_Y$. The question is how many nodes of $\mathcal{T}_X$ and $\mathcal{T}_Y$ differ. This can be answered by studying the tree $\mathsf{HT}(Z)$ built with the same dictionary $\mathfrak{D}$, where $Z \coloneqq X[i_X \mathinner{..} i_X + \ell - 1] = Y[i_Y \mathinner{..} i_Y + \ell - 1]$: On the one hand, $\mathsf{HT}(Z)$ has $\mathcal{O}(\lg^* n)$ fragile nodes on each height according to Thm. 4.23. On the other hand, each (semi-)stable node in $\mathsf{HT}(Z)$ is found in both $\mathcal{T}_X$ and $\mathcal{T}_Y$ with the same name and surname. Consequently, when traversing $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ within their respective subtrees $\mathcal{T}_X$ and $\mathcal{T}_Y$, we only visit $\mathcal{O}(\lg^* n)$ pairs of nodes per height (remember that we follow the two paths to the leaves λ_X and λ_Y, respectively, up to the point where the surnames of the visited pair of nodes match).

To sum up, we (a) compute paths from the roots to $\langle X \rangle_0[i_X]$ and $\langle Y \rangle_0[i_Y]$, respectively, in $\mathcal{O}(\lg |Y|)$ time, and (b) compare the children of at most $\mathcal{O}(\lg^* n)$ nodes per height. Since both trees have a height of $\mathcal{O}(\lg |Y|)$, we obtain our claimed running time. □

The following corollary is a small refinement of Lemma 4.25 that already shows the result of Thm. 4.3 for $\tau = 1$:

Corollary 4.26. We can endow an HSP tree of a string of length n in $\mathcal{O}(n)$ time with an $\mathcal{O}(n)$ words data structure that has the following properties: Given two HSP trees $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ built on two strings X and Y with $|X| \leq |Y| \leq n$, we can compute $\ell \coloneqq \mathsf{lcp}(X[i_X \mathinner{..}], Y[i_Y \mathinner{..}])$ in $\mathcal{O}(\lg \ell \lg^* n)$ time if both trees are endowed with this data structure, where i_X and i_Y are two text positions with $1 \leq i_X \leq |X|$ and $1 \leq i_Y \leq |Y|$.

Proof. Our idea is to endow an HSP tree with a data structure such that climbing up from a child to its parent can be performed in constant time. This can be achieved when we represent the tree topology of an HSP tree with a pointer based tree, in which each node stores its name and the pointer to its parent. The leaves are stored sequentially in a list. A bit vector with the same length

[5] We assume that $i_X + \ell \leq |X|$ and $i_Y + \ell \leq |Y|$ such that $\mathcal{T}_X$ and $\mathcal{T}_Y$ cover the mismatching pair of characters $X[i_X + \ell] \neq Y[i_Y + \ell]$. Otherwise ($i_X + \ell - 1 = |X|$ or $i_Y + \ell - 1 = |Y|$), let $\mathcal{T}_X$ and $\mathcal{T}_Y$ cover $X[i_X \mathinner{..} i_X + \ell - 1]$ and $Y[i_Y \mathinner{..} i_Y + \ell - 1]$, respectively.

as the input string is used to mark the borders of the generated substrings of the leaves. Given a text position i, we can access the leaf whose generated substring contains i in constant time with a rank-support on the bit vector. The bit vector with rank-support takes $n + o(n)$ bits. The pointer based tree can be built in $\mathcal{O}(n)$ time, and takes $\mathcal{O}(n)$ words of space. □

In the next section, we describe a preliminary version of our sparse suffix sorting algorithm that does not exploit the text space yet.

4.4 Sparse Suffix Sorting

The sparse suffix sorting problem asks for the order of suffixes starting at certain positions in a text T. In our case, these positions only need be given online, i.e., sequentially and in an arbitrary order. We collect them conceptually in a dynamic set $\mathcal{P}$ with $m \coloneqq |\mathcal{P}|$. The online sparse suffix sorting problem is to keep the suffixes starting at the positions stored in the incrementally growing set $\mathcal{P}$ in sorted order. Due to the online setting, we represent the order of $\mathit{Suf}(\mathcal{P})$ by a dynamic, self-balancing binary search tree (e.g., an AVL tree). Each node of the tree is associated with a distinct suffix in $\mathit{Suf}(\mathcal{P})$; the lexicographic order is used as the sorting criterion.

The technique of Irving and Love [134] augments an AVL tree on a set of strings $\mathcal{S}$ with the lengths of LCPs so that $\ell_Y \coloneqq \max\{\mathsf{lcp}(X, Y) \mid X \in \mathcal{S}\}$ can be computed in $\mathcal{O}(\ell_Y / \log_\sigma n + \lg |\mathcal{S}|)$ time for a string Y, where the division by $\log_\sigma n$ is due to the word-packing technique (recall Sect. 2.7). Inserting a new string Y into the tree is supported in the same time complexity (ℓ_Y is defined as before). Irving and Love called this data structure the *suffix AVL tree* on $\mathcal{S}$; we denote it by $\mathsf{SAVL}(\mathcal{S})$.

Remembering Sect. 4.1, our goal is to build $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ efficiently. However, inserting m suffixes naïvely takes $\Omega(|\mathcal{C}|\, m / \log_\sigma n + m \lg m)$ time. How to speed up the comparisons by exploiting a data structure for LCE queries is the topic of this section.

4.4.1 Abstract Algorithm

Starting with an empty set of positions $\mathcal{P} = \emptyset$, our algorithm incrementally updates $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ on the input of every new text position, involving LCE computations between the new suffix and suffixes already stored in $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$. A crucial part of the algorithm is performed by these LCE computations, for which an LCE data structure is advantageous to have. In particular, we are interested in a *mergeable* LCE data structure that is mergeable in such a way that the merged instance answers queries faster than performing a query on both former instances separately. We call this a *dynamic LCE data structure (dynLCE)*; it supports the following operations:

- $\mathsf{dynLCE}(\mathcal{I})$ constructs a dynLCE data structure M on the substring $T[\mathcal{I}]$. Let $M.\mathsf{ival}$ denote the interval $\mathcal{I}$.
- $\mathsf{lce}(M_1, M_2, p_1, p_2)$ computes $\mathsf{lce}(p_1, p_2)$, where $p_i \in M_i.\mathsf{ival}$ for $i = 1, 2$.
- $\mathsf{merge}(M_1, M_2)$ merges two dynLCEs M_1 and M_2 such that the output is a dynLCE built on the string concatenation of $T[M_1.\mathsf{ival}]$ and $T[M_2.\mathsf{ival}]$.

We use the expression $t_{\mathrm{C}}(|\mathcal{I}|)$ to denote the construction time of such a data structure on the substring $T[\mathcal{I}]$. We assume that the construction of $\mathsf{dynLCE}(\mathcal{I})$ takes at least as long as scanning all characters on Y, i.e.,

Property 1: $t_{\mathrm{C}}(|\mathcal{I}|) = \Omega(|\mathcal{I}| \,/\, \log_\sigma n)$.

We use the expressions $t_{\mathrm{Q}}(|X|+|Y|)$ and $t_{\mathrm{M}}(|X|+|Y|)$ to denote the time for querying and the time for merging two such data structures built on two given strings X and Y, respectively. Querying two dynLCEs for a length ℓ is at least as fast as the word-packed character comparison if and only if $\ell = \Omega(t_{\mathrm{Q}}(\ell) \log_\sigma n)$.[6] Hence, we obtain the following property:

Property 2: A dynLCE on a text smaller than $g := \Theta(t_{\mathrm{Q}}(g) \log_\sigma n)$ is always slower than the word-packed character comparison.

In the following, we build dynLCEs on substrings of the text. Each interval of the text that is covered by a dynLCE is called an *LCE interval*. The LCE intervals are maintained in a self-balancing binary search tree $\mathcal{L}$ of size $\mathcal{O}(m)$. The tree $\mathcal{L}$ stores the starting and the ending positions of each LCE interval, and uses the starting positions as keys to answer the queries

- whether a position is covered by a dynLCE, and
- where the next text position starts that is covered by a dynLCE,

in $\mathcal{O}(\lg m)$ time. Additionally, each LCE interval is assigned to one dynLCE data structure (a dynLCE can be assigned to multiple LCE intervals) such that $\mathcal{L}$ cannot only retrieve the next position covered by a dynLCE, but actually return a dynLCE that covers that position. The dynLCE is retrieved by augmenting an LCE interval $\mathcal{I}$ with a pointer to its dynLCE data structure M, and with an integer i such that $T[M.\mathsf{ival} \cap [i \,..\, i + |\mathcal{I}| - 1]] = T[\mathcal{I}]$ (since M could be built on a text interval $M.\mathsf{ival} \neq \mathcal{I}$ that contains an occurrence of $T[\mathcal{I}]$).

Given a new position $\hat{p} \notin \mathcal{P}$ with $1 \leq \hat{p} \leq |T|$, updating $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ to $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P} \cup \{\hat{p}\}))$ involves two parts: first *locating* the insertion node for $\hat{p}$ in $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$, and then *updating* the set of LCE intervals. After processing these two parts we insert $\hat{p}$ in $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$.

[6] We assume that $t_{\mathrm{Q}}(\ell)$ is sub-linear in ℓ.

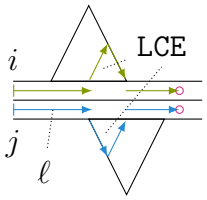
Algorithm 15: Hybrid LCE Algorithm – Locating Process. Let $\mathcal{I}.\mathsf{ref}$ be the dynLCE of an LCE interval $\mathcal{I}$.

```
input : text positions p, p' ∈ P,
        search tree L maintaining all LCE intervals
1  function lce(p, p')          ▷ called by Algo. 16 to find the insertion position for p̂
2      I ← L.predecessor(p) and J ← L.predecessor(p')
                                  ▷ I = argmax {b(I) | I ∈ L ∧ b(I) < p}
3      if e(I) > p and e(J) > p' then
4          ℓ ← lce(I.ref, J.ref, p, p')   ▷ LCE with dynLCEs built on I and J
5      else   ▷ compare naïvely until reaching positions covered by LCE intervals (the
           fourth argument of naïve_compare is the number of characters to compare at
           most)
6          I ← L.successor(p) and J ← L.successor(p')
                                  ▷ I = argmin {b(I) | I ∈ L ∧ b(I) > p}
7          ℓ ← naïve_compare(T, p, p', max(b(I) − p, b(J) − p'))
8      p ← p + ℓ and p' ← p' + ℓ
9      if T[p] ≠ T[p'] then return ℓ                    ▷ found mismatch
10     return ℓ + lce(p, p')   ▷ recurse since we reached either the beginning or the end
           of an LCE interval covered by L
```

Locating. The insertion operation performs an LCE computation for each node encountered in $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ while locating the insertion point of $\hat{p}$. Suppose that the task is to compare the suffixes $T[i\,..]$ and $T[j\,..]$ for two text positions i and j with $1 \leq i, j \leq |T|$. We perform the following steps to compute $\mathsf{lce}(i, j)$:

(1) Check whether the positions i and j are contained in an LCE interval, in $\mathcal{O}(\lg m)$ time with the search tree $\mathcal{L}$.

- If both positions are covered by LCE intervals, then query the respective dynLCEs for the length ℓ of the LCE starting at i and j. Increment i and j by ℓ. Return the number of compared characters on finding a mismatch while computing the LCE.
- Otherwise (if i or j are not contained in an LCE interval), find the smallest length ℓ such that $i + \ell$ and $j + \ell$ are covered by LCE intervals. Increment i and j by ℓ, and naïvely compare ℓ characters. Return the number of compared characters on a mismatch.



(2) Return the total number of matched positions if a mismatch is found in (1). Otherwise, repeat the above check again (with the incremented values of i and j).

Algorithm 16: Sparse Suffix Sorting

input : search tree $\mathcal{L}$ maintaining all LCE intervals,
suffix AVL tree $S := \mathsf{SAVL}(Suf(\mathcal{P}))$,
text position $\hat{p} \in [1\,..\,n] \setminus \mathcal{P}$

1 function *add*($\hat{p}$) ▷ online query with a text position $\hat{p}$
2 $v \leftarrow S.\mathsf{root}$ ▷ locate insertion position of $T[\hat{p}\,..]$ in S
3 while v is not a leaf do $v \leftarrow S.\mathsf{locate\_child}(v, \hat{p}, \mathsf{lce})$ ▷ perform on each depth an LCE query with a suffix, use Algo. 15
4 $\bar{p} \leftarrow \mathrm{mlcparg}(\hat{p})$ and $\ell \leftarrow \mathsf{lce}(\bar{p}, \hat{p})$ ▷ already computed above
5 $S.\mathsf{insert}(v, \hat{p})$ ▷ add the suffix $T[\hat{p}\,..]$ to S
6 if $\ell < 2g$ then return ▷ Property 3
7 $A \leftarrow \{\mathcal{I} \in \mathcal{L} \mid \mathcal{I} \cap ([\hat{p}-g\,..\,\hat{p}+g+\ell-1] \cup [\bar{p}-g\,..\,\bar{p}+g+\ell-1]) \neq \emptyset\}$
8 $U \leftarrow \bigcup_{\mathcal{I}\in A} \mathcal{I}$ ▷ union of the intervals that are interesting for merging
9 for each interval $\mathcal{I} \in ([\hat{p}\,..\,\hat{p}+\ell-1] \cup [\bar{p}\,..\,\bar{p}+\ell-1]) \setminus U$ do
either
10 • assign $\mathcal{I}.\mathsf{ref}$ to a dynLCE in case we found an appropriate dynLCE while applying Rule 1 or 2, or
• set $\mathcal{I}.\mathsf{ref} \leftarrow \mathsf{dynLCE}(T[\mathcal{I}])$ otherwise according to Rule 3
11 $\mathcal{L}.\mathsf{insert}(\mathcal{I})$ and $A.\mathsf{insert}(\mathcal{I})$
12 sort A ascendingly by the starting positions of its intervals
13 $\mathcal{I} := A.\mathsf{pop}()$ ▷ remove first element from the queue A and return it
14 while A is not empty do ▷ enforce Property 5
15 $\mathcal{J} \leftarrow A.\mathsf{pop}()$
16 if $\mathsf{b}(\mathcal{J}) - \mathsf{e}(\mathcal{I}) \leq g$ then ▷ Property 5 violated
17 $\mathcal{I}' \leftarrow \mathcal{I} \cup \mathcal{J}$ ▷ merge both intervals
18 $\mathcal{I}'.\mathsf{ref} \leftarrow \mathsf{merge}(\mathcal{I}.\mathsf{ref}, \mathcal{J}.\mathsf{ref})$ ▷ apply Rule 4
19 $\mathcal{L}.\mathsf{delete}(\mathcal{I})$ and $\mathcal{L}.\mathsf{delete}(\mathcal{J})$
20 $\mathcal{L}.\mathsf{insert}(\mathcal{I}')$
21 $\mathcal{I} \leftarrow \mathcal{I}'$
22 $\mathcal{P} \leftarrow \mathcal{P} \cup \{\hat{p}\}$

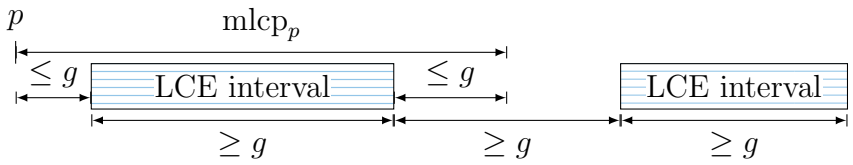


Fig. 4.27: Sketch of two LCE intervals with Properties 3 to 5.

The steps are additionally listed in Algo. 15. After locating the insertion point of $\hat{p}$ in $\mathsf{SAVL}(Suf(\mathcal{P}))$, we obtain

$$\bar{p} \coloneqq \text{mlcparg}(\hat{p}) \text{ and } \ell \coloneqq \text{mlcp}(\hat{p})$$

as a byproduct, where

$$\text{mlcparg}(p) \coloneqq \text{argmax}_{p' \in \mathcal{P}, p \neq p'} \mathsf{lcp}(T[p\,..], T[p'\,..])$$

and

$$\text{mlcp}(p) \coloneqq \mathsf{lcp}(T[p\,..], T[\text{mlcparg}(p)\,..])$$

for each text position p with $1 \leq p \leq |T|$. We insert $\hat{p}$ into $\mathsf{SAVL}(Suf(\mathcal{P}))$, and use the position $\bar{p}$ and the length ℓ to update the LCE intervals.

Updating. The LCE intervals are updated dynamically, subject to the following properties (see Fig. 4.27):

Property 3: The length of each LCE interval is at least g (defined in Property 2).

Property 4: For every $p \in \mathcal{P}$, the interval $[p\,..\,p+\text{mlcp}(p)-1]$ is covered by an LCE interval, *except at most* g positions at its left and right ends.

Property 5: There is a gap of at least g positions between every pair of LCE intervals.

After adding $\hat{p}$ to $\mathcal{P}$, we perform the following instructions to satisfy the properties. If $\ell \leq 2g$, we do nothing, because all properties are still valid (in particular, Property 4 still holds). Otherwise, we need to restore Property 4. There are at most two positions in $\mathcal{P}$ that possibly invalidate Property 4 after adding $\hat{p}$, and these are $\hat{p}$ and $\bar{p}$ (otherwise, by transitivity, we would have created a longer LCE interval previously).

We introduce an algorithm that does not restore Property 4 directly, but first ensures that

Property 4^+: the intervals $[\hat{p}\,..\,\hat{p}+\ell-1]$ and $[\bar{p}\,..\,\bar{p}+\ell-1]$ are covered by one or multiple LCE intervals.

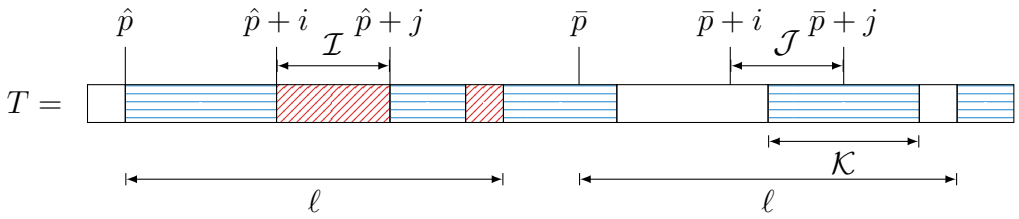


Fig. 4.28: Application of Rules 1 to 4 for preserving the properties. The interval $\mathcal{I} \coloneqq [\hat{p}+i \,..\, \hat{p}+j]$ is not yet covered by an LCE interval, but is contained in $[\hat{p} \,..\, \hat{p}+\ell-1]$ — a conflict with Property 4. The conflict is resolved based on the LCE intervals covering the positions of $\mathcal{J} \coloneqq [\bar{p}+i \,..\, \bar{p}+j]$. The intervals with the blue horizontal lines (▤) are the LCE intervals, and the intervals with the diagonal red lines (▨) are the intervals of $[\hat{p} \,..\, \hat{p}+\ell-1] \setminus U$. Here, $\mathcal{J}$ intersects with an LCE interval $\mathcal{K}$. This case is treated in Rule 2.

In the following, we first process the LCE intervals to satisfy Property 4^+ and then subsequently to satisfy Property 5. When Property 4^+ and Property 5 are satisfied, then Property 4 is also satisfied. We can satisfy Property 4^+ with the following steps: Let $U \subset [1 \,..\, n]$ be the set of all positions that belong to an LCE interval. The set $[\hat{p} \,..\, \hat{p}+\ell-1] \setminus U$ can be represented as a set of disjoint intervals of maximal length. For each interval $\mathcal{I} \coloneqq [\hat{p}+i \,..\, \hat{p}+j] \subset [\hat{p} \,..\, \hat{p}+\ell-1]$ of that set, apply the following rules with $\mathcal{J} \coloneqq [\bar{p}+i \,..\, \bar{p}+j]$ (for integers i, j with $0 \le i \le j \le \ell-1$, see Fig. 4.28) sequentially (cf. Line 9 of Algo. 16):

Rule 1: If $\mathcal{J}$ is a sub-interval of an LCE interval $\mathcal{K}$, then declare $\mathcal{I}$ as an LCE interval and let it refer to the dynLCE of $\mathcal{K}$.

Rule 2: If $\mathcal{J}$ intersects with an LCE interval $\mathcal{K}$, enlarge the dynLCE on $T[\mathcal{K}]$ to cover $T[\mathcal{K} \cup \mathcal{J}]$ (create a dynLCE on $T[\mathcal{J} \setminus \mathcal{K}]$ and merge it with the dynLCE on $T[\mathcal{K}]$). Apply Rule 1.

Rule 3: Otherwise (there is no LCE interval $\mathcal{K}$ with $\mathcal{J} \cap \mathcal{K} \neq \emptyset$), create $\mathsf{dynLCE}(\mathcal{J})$, and make $\mathcal{I}$ and $\mathcal{J}$ to LCE intervals referring to $\mathsf{dynLCE}(\mathcal{J})$.

We satisfy Property 4^+ on $[\bar{p} \,..\, \bar{p}+\ell-1]$ by updating U, computing the set of disjoint intervals $[\bar{p} \,..\, \bar{p}+\ell-1] \setminus U$, and applying the same rules on it. However, Rule 1 or Rule 3 can create LCE intervals shorter than g, violating Property 3. By construction, such a short LCE interval is adjacent to another LCE interval (the rules compute a cover of $[\hat{p} \,..\, \hat{p}+\ell-1]$ and $[\bar{p} \,..\, \bar{p}+\ell-1]$ with LCE intervals). This means that we can restore Property 3 by restoring Property 5. We do that by applying the following rule subsequently to Rule 3:

Rule 4: Merge a newly created or merged LCE interval violating Property 5 with its nearest LCE interval (ties can be broken arbitrarily). Recurse until no merge occurs (cf. Line 14 of Algo. 16).

Rule 4 finally restores Property 4 (since Property 4^+ and Property 5 hold). After applying all rules, we have introduced at most two[7] new LCE intervals that cover the intervals $[\hat{p}+g\,..\,\hat{p}+\ell-1-g]$ and $[\bar{p}+g\,..\,\bar{p}+\ell-1-g]$, respectively, to satisfy Properties 3 to 5. We summarize the steps taken for adding $\hat{p}$ to $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ in Algo. 16. The running time of this algorithm is analyzed in the following lemma:

Lemma 4.27. Given a text T of length n and a set of m arbitrary positions $\mathcal{P}$ in T, the suffix AVL tree $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ with the suffixes of T starting at the positions $\mathcal{P}$ can be computed deterministically in $\mathcal{O}(t_{\mathrm{C}}(|\mathcal{C}|)+t_{\mathrm{Q}}(|\mathcal{C}|)m\lg m+t_{\mathrm{M}}(|\mathcal{C}|)m)$ time.

Proof. The analysis is split into managing the dynLCEs, and the LCE queries:

- We build dynLCEs on substrings covering at most $|\mathcal{C}|$ characters of the text, taking at most $t_{\mathrm{C}}(|\mathcal{C}|)$ time for constructing all dynLCEs. During the construction of the dynLCEs we spend $\mathcal{O}(|\mathcal{C}|\,/\log_\sigma n)=\mathcal{O}(t_{\mathrm{C}}(|\mathcal{C}|))$ time on character comparisons due to Property 1.
- The number of merge operations on the LCE intervals is upper bounded by $2m$ in total, since we create at most two new LCE intervals for every position in $\mathcal{P}$. In total, we spend at most $2\,t_{\mathrm{M}}(|\mathcal{C}|)m$ time for the merging.
- The algorithm performs $\mathcal{O}(m\lg m)$ LCE queries. LCE queries involve either (a) character comparisons or (b) querying a dynLCE. Given that we have $\delta<2m$ LCE intervals, we switch between both techniques at most $4\delta+1$ times for an LCE query.
  (a) On the one hand, the overall time for the character comparisons is bounded by $\mathcal{O}(t_{\mathrm{C}}(|\mathcal{C}|)+t_{\mathrm{Q}}(|\mathcal{C}|)m\lg m)$:
    - By Property 3, all substrings $T[p\,..\,p+\mathrm{mlcp}(p)-1]$ are covered by an LCE interval, except at most at $2g$ positions. This means that all substrings that are not covered by an LCE interval, but have been subject to a character comparison, are shorter than $2g$. For a character comparison with one of those substrings, we spend at most $\mathcal{O}(gm\lg m/\log_\sigma n)=\mathcal{O}(t_{\mathrm{Q}}(g)m\lg m)=\mathcal{O}(t_{\mathrm{Q}}(|\mathcal{C}|)m\lg m)$ time. In the case that $g>|\mathcal{C}|$, we do not create any LCE interval, and spend $\mathcal{O}(gm\lg m/\log_\sigma n)=\mathcal{O}(t_{\mathrm{Q}}(|\mathcal{C}|)m\lg m)$ overall time due to Property 2.

[7] The number of new LCE intervals could be indeed two: Although $\bar{p}\in\mathcal{P}$, we would not have created an LCE interval covering $[\bar{p}+g\,..\,\bar{p}+\bar{\ell}-1-g]$ if $\mathrm{mlcp}(\bar{p})$ was smaller than g at the time when we inserted $\bar{p}$ in $\mathcal{P}$ with $\bar{\ell}:=\mathrm{mlcp}(\bar{p})$.

– If we compare more than g characters for an LCE query, we create at most two LCE intervals, possibly involving the construction of dynLCEs on the compared substrings. The construction of a dynLCE on an interval $\mathcal{I}$ takes $t_C(|\mathcal{I}|) = \Omega(|\mathcal{I}| / \log_\sigma n)$ time due to Property 1. Hence, the time needed for character comparisons is $\mathcal{O}(|\mathcal{I}| / \log_\sigma n) = \mathcal{O}(t_C(|\mathcal{I}|))$. This sums up to $\mathcal{O}(t_C(|\mathcal{C}|))$ total time spent on character comparisons of substrings longer than g characters.

(b) On the other hand, querying the dynLCEs take at most $\mathcal{O}(t_Q(|\mathcal{C}|)m \lg m)$ overall time. Suppose that we look up $d < \delta$ LCE intervals for an LCE query. Since we look up an LCE interval in $\mathcal{O}(\lg m)$ time with $\mathcal{L}$, we spend $\mathcal{O}(d \lg m)$ time on the lookups during this LCE query. However, we subsequently merge all d looked-up LCE intervals, reducing the number of LCE intervals δ by $d - 1$. Consequently, we perform a look-up of an LCE interval at most $2m$ times in total. □

The last step is to compute $\mathsf{SSA} := \mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP} := \mathsf{SLCP}(T, \mathcal{P})$ from $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ by traversing $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ and performing LCE queries on the already computed dynLCEs: The $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ is a binary search tree storing all elements of $\mathit{Suf}(\mathcal{P})$ in lexicographically sorted order. Consequently, we can compute SSA with an in-order traversal of $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$. Afterwards, we compute $\mathsf{SLCP}[i] = \mathsf{lce}(\mathsf{SSA}[i], \mathsf{SSA}[i-1])$. If the text positions $[\mathsf{SSA}[i] \,..\, \mathsf{SSA}[i] + \mathsf{SLCP}[i] - 1]]$ and $[\mathsf{SSA}[i-1] \,..\, \mathsf{SSA}[i-1] + \mathsf{SLCP}[i] - 1]]$ are not covered by an LCE interval, then $\mathsf{SLCP}[i] = \mathcal{O}(g)$ due to Property 3, and we spend at most $\mathcal{O}(g / \log_\sigma n)$ time on computing $\mathsf{SLCP}[i]$ by character comparisons. Otherwise, we spend $\mathcal{O}(g / \log_\sigma n + t_Q(\mathsf{SLCP}[i])) = \mathcal{O}(t_Q(\mathsf{SLCP}[i]))$ time by querying a *single* dynLCE due to Property 4. Querying whether both text intervals are covered by an dynLCE costs $\mathcal{O}(\lg m)$ time with $\mathcal{L}$. In total, we can compute $\mathsf{SLCP}[i]$ for each integer i with $2 \leq i \leq m$ in $\mathcal{O}(t_Q(|\mathcal{C}|)m \lg m)$ time, since $\mathcal{O}(g / \log_\sigma n) = \mathcal{O}(t_Q(g))$ due to Property 2. The following corollary of Lemma 4.27 summarizes the achievements of this section:

Corollary 4.28. Given a text T of length n that is loaded into RAM, the SSA and SLCP of T for a set of m arbitrary positions can be computed deterministically in $\mathcal{O}(t_C(|\mathcal{C}|) + t_Q(|\mathcal{C}|)m \lg m + t_M(|\mathcal{C}|)m)$ time. We need $\mathcal{O}(m)$ words of space, and the space to store instances of dynLCE on $|\mathcal{C}|$ positions.

4.4.2 Sparse Suffix Sorting with HSP Trees

We show that the HSP tree is a dynLCE data structure. Remembering that the algorithm from Sect. 4.4.1 depends on the merging operation of dynLCE, we now introduce the merging of HSP trees. A naïve way to merge two HSP trees $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ is to build $\mathsf{HT}(XY)$ completely from scratch. Since only

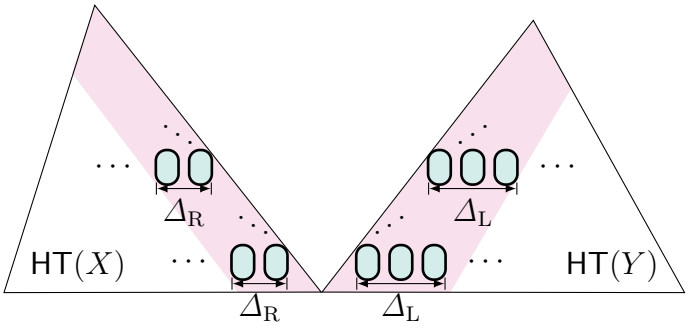


Fig. 4.29: Merging $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$. Given that both trees contain only Type 2 nodes, it suffices to apply the parsing on the Δ_R rightmost nodes and Δ_L leftmost nodes of $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$, respectively, to obtain $\mathsf{HT}(XY)$.

the fragile nodes of $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ can change when merging both trees, a more sophisticated approach would reparse only the fragile nodes of both trees. Remembering the properties studied in Sect. 4.2.4, we show such an approach in the following lemma:

Lemma 4.29. Merging $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ of two strings $X, Y \in \Sigma^*$ into $\mathsf{HT}(XY)$ takes $\mathcal{O}(t_\mathrm{look}(\Delta_\mathrm{R} \lg |X| + \Delta_\mathrm{L} \lg |Y|))$ time.

Proof. First assume that $\mathsf{HT}(X)$ and $\mathsf{HT}(Y)$ only contain Type 2 nodes. In this case, we examine the rightmost nodes of $\mathsf{HT}(X)$ and the leftmost nodes of $\mathsf{HT}(Y)$ from the bottom up to the root: At each height h, we merge the nodes $\langle X \rangle_h$ and $\langle Y \rangle_h$ to $\langle XY \rangle_h$ by reparsing the Δ_R rightmost nodes of $\langle X \rangle_h$, and the Δ_L leftmost nodes of $\langle Y \rangle_h$ (see Fig. 4.29). By doing so, we reparse all nodes of $\mathsf{HT}(X)$ (resp. $\mathsf{HT}(Y)$) whose local surrounding on the right (resp. left) side does not exist. Nodes of $\mathsf{HT}(X)$ (resp. $\mathsf{HT}(Y)$) that have a local surrounding on the right (resp. left) side are not changed by the parsing. In total, we spend $\mathcal{O}(t_\mathrm{look}(\Delta_\mathrm{R} \lg |X| + \Delta_\mathrm{L} \lg |Y|))$ time on merging two trees consisting of Type 2 nodes.

Next, we allow repeating nodes. Lemma 4.21 shows that there are no fragile surrounded nodes in $\mathsf{HT}(X)$ that need to be fixed. The remaining problem is to find and recompute the surrounded nodes in $\mathsf{HT}(Y)$ whose names change on merging both trees. The lowest of these nodes belong to a repeating meta-block due to Lemma 4.7 and Cor. 4.16. To find this meta-block, we adapt the strategy of the first paragraph considering only Type 2 meta-blocks. On each height h, we reparse the Δ_L leftmost nodes of $\langle Y \rangle_h$. If the rightmost of these Δ_L nodes are contained in a repeating meta-block μ that does not end within those Δ_L leftmost nodes, chances are that the names of some nodes in μ change. Due to Cor. 4.16, it is sufficient to reparse the two rightmost nodes of μ. This is done as follows (cf. Fig. 4.30):

1. Take the leftmost repetitive node s of μ (which exists due to Cor. 4.20, and is one of the $\Delta_\mathrm{L} + 1$ leftmost nodes on height h).

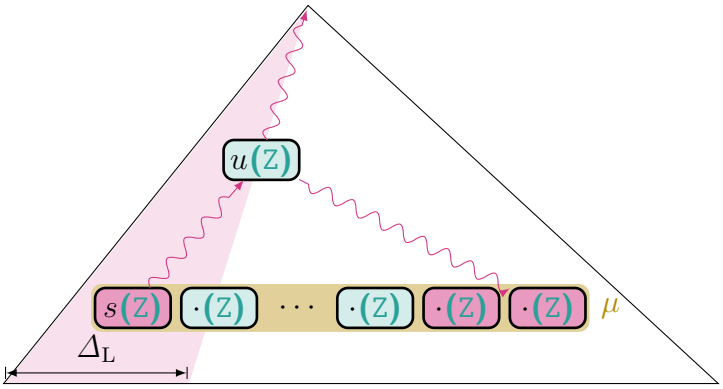


Fig. 4.30: Reparsing of fragile surrounded nodes during a merging operation. The lowest fragile surrounded nodes that need to be reparsed belong to a meta-block μ whose leftmost nodes are non-surrounded. Given that the repetitive nodes of μ have the surname Z, we can access all fragile nodes of μ by climbing up from the leftmost repetitive node s of μ to the highest node u with surname Z, and subsequently descending down to the rightmost repetitive nodes of μ in $\mathcal{O}(\lg|\mu|)$ time.

2. Given that s has the surname Z, climb up the tree to find the highest ancestor u with surname Z. The ancestor u is the LCA of s and the rightmost repetitive node of μ.

3. Walk down from u to the rightmost nodes of μ.

4. Reparse μ's two rightmost nodes.

5. Reparse all ancestors of these two nodes that are surrounded.

6. Check whether the reparsed ancestors invalidate the parsing of their meta-blocks; fix the parsing for those meta-blocks recursively.

Climbing up to find u and walking down to the rightmost nodes of μ takes $\mathcal{O}(t_{\text{look}}\lg|\mu|) = \mathcal{O}(t_{\text{look}}\lg(n/2^h))$ time, reparsing the surrounded ancestor nodes of the two rightmost nodes of μ takes $\mathcal{O}(t_{\text{look}}\lg(n/2^h))$ time. Given that the highest nodes of this reparsing are on a height $h' > h$, Lemma 4.22 states that up to the height $h'+1$, there is no need to reparse a fragile surrounded node (we follow the paths of fragile nodes as depicted in Fig. 4.25). Given that there are $\mu_1, \ldots, \mu_k$ such meta-blocks (for which we apply Steps 1 to 6), we have $\mathcal{O}(t_{\text{look}}\sum_{i=1}^{k}\lg|\mu_i|) = \mathcal{O}(t_{\text{look}}\lg n)$ due to $\sum_{i=1}^{k}\lg|\mu_i| \leq \lg n$. Hence, we spend $\mathcal{O}((\Delta_{\text{L}}+\Delta_{\text{R}})t_{\text{look}}\lg|Y|)$ time overall. □

The following theorem combines the results of Cor. 4.28 and Lemma 4.29.

Theorem 4.30. Given a text T of length n and a set of m text positions $\mathcal{P}$, $\mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP}(T, \mathcal{P})$ can be computed in $\mathcal{O}(|\mathcal{C}|\,(\lg^* n + t_{\text{look}}) + m \lg m \lg n \lg^* n)$ time. We need $\mathcal{O}(n + m)$ words of space.

Proof. We have

- $t_{\text{C}}(|\mathcal{C}|) = \mathcal{O}(|\mathcal{C}|\,(\lg^* n + t_{\text{look}}))$ due to Lemma 4.24,
- $t_{\text{Q}}(|\mathcal{C}|) = \mathcal{O}(\lg^* n \lg n)$ due to Lemma 4.25, and
- $t_{\text{M}}(|\mathcal{C}|) = \mathcal{O}(t_{\text{look}} \lg n \lg^* n)$ due to Lemma 4.29.

Actually, the time cost for merging is already upper bounded by the cost for the tree creation. To see this, let $\delta \leq m$ be the number of LCE intervals. Since each HSP tree covers at least g characters, δg is at most $|\mathcal{C}|$, and we obtain $\delta\, t_{\text{M}}(|\mathcal{C}|) = \mathcal{O}(|\mathcal{C}|\, t_{\text{M}}(|\mathcal{C}|)/g) = \mathcal{O}(|\mathcal{C}|\, t_{\text{look}})$ overall time for merging, where $g = \Theta(t_{\text{Q}}(|\mathcal{C}|) \lg n / \lg \sigma) = \Theta(\lg^* n \lg^2 n / \lg \sigma)$. Plugging the times $t_{\text{C}}(|\mathcal{C}|)$, $t_{\text{Q}}(|\mathcal{C}|)$, and the refined analysis of the merging time cost in Cor. 4.28 yields the claimed time bounds. □

4.5 Sparse Suffix Sorting in Text Space

Remembering the outline in the introduction, the key idea to solve the limited space problem is storing dynLCEs in text space. Taking two LCE intervals of the text containing the same substring, we free up the space of *one* part while marking the *other* part as a reference. The freed space could be used to store an HSP tree whose leaves refer to substrings of the other LCE interval. By doing so, we would use the text space for storing the HSP trees, while using only $\mathcal{O}(m)$ additional words for storing $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ and the search tree $\mathcal{L}$ of the LCE intervals. However, an HSP tree built on a string of length n takes $\mathcal{O}(n \lg n)$ bits, while the string itself provides only $n \lg \sigma$ bits. Our solution is to truncate the HSP tree at a fixed height η, discarding the nodes in the lower part. The truncated version $\mathsf{tHT}_\eta(Y)$ stores just the upper part, while its new leaves refer to (possibly long) substrings of Y. The resulting tree is called the *η-truncated HSP tree* (tHT_η), whose definition follows:

4.5.1 Truncated HSP Trees

We define a height η and delete all nodes at heights less than η, which we call *lower nodes*. A node higher than η is called an *upper node*. The nodes at height η form the new leaves and are called *η-nodes*. Similar to the former leaves, their names are pointers to their generated substrings appearing in Y. Remembering that each internal node has two or three children, an η-node generates a string of length at least 2^η and at most 3^η. The maximum number of nodes in an

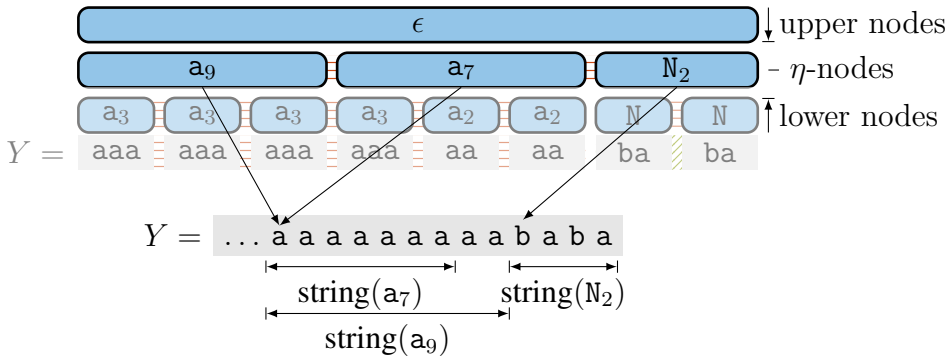


Fig. 4.31: The η-truncated HSP tree $\mathsf{tHT}_\eta(Y)$ of the substring Y defined in Fig. 4.7 with $\eta = 2$. Like in Fig. 4.47, the lower nodes are grayed out. An η-node is a leaf in $\mathsf{tHT}_\eta(Y)$, and has a generated substring with a length between four and nine.

η-truncated HSP tree of a string of length n is $n/2^\eta$. Figure 4.31 shows an example with $\eta = 2$.

Similar to leaves in untruncated HSP trees, we use the generated substring X of an η-node v for storing and looking up v: While the leaves of the HSP tree have a generated substring of constant size (two or three characters), the generated substring of an η-node can be as long as 3^η. Storing such long strings in a binary search tree representing the reverse dictionary of $\mathfrak{D}$ is inefficient; it would need $\mathcal{O}(\ell \lg \sigma)$ time for a lookup or insertion of a key of length ℓ. Instead, we want a dictionary data structure storing $\mathcal{O}(|Y|)$ elements in $\mathcal{O}(|Y|)$ words of space[8], supporting lookup and insert in $\mathcal{O}(t_{\text{look}} + \ell/\log_\sigma n)$ time for a key of length ℓ. For instance, Franceschini and Grossi's data structure [94] with word-packing supports the desired time and space bounds with $t_{\text{look}} = \mathcal{O}(\lg n)$.

Lemma 4.31. We can build an η-truncated HSP tree $\mathsf{tHT}_\eta(Y)$ of a string Y of length n in $\mathcal{O}(n(\lg^* n + \eta/\log_\sigma n + t_{\text{look}}/2^\eta))$ time, using $\mathcal{O}(3^\eta \lg^* n)$ words of working space. The tree takes $\mathcal{O}(n/2^\eta)$ words of space.

Proof. Instead of building the HSP tree level by level, we compute the η-nodes one after another, from left to right. We can split the parsing of the whole string into several parts. Each part computes one η-node.

First assume that $\mathsf{tHT}_\eta(Y)$ only contains Type 2 nodes. Then the name of an η-node v is determined by v's local surrounding (as far as it exists) due to Lemma 4.7. Thus it is sufficient to keep v's local surrounding at height $\eta - 1$, which we denote by X_v, in memory. X_v is a string of lower nodes. To parse a string of lower nodes by HSP, we have to give each lower node a name. Unfortunately, storing the names of all lower nodes in a dictionary would take too much space. Instead, we create the name of a lower node temporarily by

[8] The data structure is not necessarily stored in consecutive space like an array.

setting the name of a lower node to its generated substring. A drawback is that we cannot retrieve their names later. Luckily, we only need the names of the lower nodes for constructing X_v. We construct X_v as follows: Given that we parsed the local surrounding of v at height h ($0 \leq h \leq \eta - 3$) with HSP, we store the borders of the blocks on height $h+1$ in an integer array such that we can access the name (i.e., the generated substring) of the i-th block on height $h+1$. With this integer array, we can parse the blocks on height $h+1$ to obtain the blocks on height $h+2$, whose borders are again stored in an integer array. Having the borders of the blocks on height $h+2$, we can remove the integer array on height $h+1$. The blocks on height $\eta - 1$ are the nodes of X_v.

In the general case (when $\mathsf{tHT}_\eta(Y)$ contains repeating nodes), it can happen that the name of a greedily parsed node (i.e., a repeating node or one of the Δ_L leftmost nodes of a Type 2 meta-block) depends not necessarily on its local surrounding, but on the length of its repeating meta-block, its surname and its children (in case of a Type M node). This means that when computing X_v of an η-node v, we additionally have to consider the case when nodes in the local surrounding of v are contained in a meta-block μ on height $h < \eta$ that extends over the nodes in v's surrounding at height h. It is sufficient to use a counting variable that tracks the position of the last block of μ belonging to the subtree of the preceding η-node of v (remember that the greedy parsing determines the blocks by an arithmetic progression, cf. Fig. 4.19). Another necessity is to maintain the surnames of the lower nodes. In our approach, each array storing the borders of the blocks on the heights below η is accompanied with two arrays. The first array stores the length of the prefix of the generated substring of each block β that is equal to β's surname; the second array stores the surname-length of each block.

Working Space. We construct v after constructing X_v. To construct X_v, we apply the HSP technique $(\eta - 1)$ times on the generated substring of the nodes in X_v. Since the nodes of X_v cover at most $3^\eta(\Delta_\mathrm{L} + \Delta_\mathrm{R})$ characters, we need $\mathcal{O}(3^\eta(\Delta_\mathrm{L} + \Delta_\mathrm{R}))$ words of working space to maintain the integer arrays storing the borders of the blocks at two consecutive heights. To cope with the meta-blocks extending over the border of the subtrees of two η-nodes, we store the last position of each such meta-block belonging to the local surrounding of the previous η-node. These positions take $\mathcal{O}(\eta)$ words, since such a meta-block can exist on every height below η.

Time. The time bound $\mathcal{O}(n \lg^* n)$ for the repeated application of the alphabet reduction is the same as in Lemma 4.24. The new part is the construction of an η-node by constructing X_v: To construct the lower nodes X_v, we apply the HSP technique $(\eta - 1)$ times on $\mathsf{string}(v)$. The HSP technique compares lower nodes by their generated substrings (instead of comparing by a name stored in $\mathfrak{D}$). It always compares two adjacent lower nodes during the construction

of X_v. To bound the number of comparisons of the lower nodes, we focus on all lower nodes on a fixed height h with $1 \leq h \leq \eta - 1$: Since the sum of the lengths of the generated substrings of the lower nodes on height h is always n, the comparisons of the lower nodes on height h take $\mathcal{O}(n/\log_\sigma n)$ time, independent of the number of nodes on height h. Summing over all heights, these comparisons take $\mathcal{O}(n\eta/\log_\sigma n)$ time in total. By the same argument, maintaining the names of all η-nodes takes $\mathcal{O}(n/\log_\sigma n + t_{\text{look}} n/2^\eta)$ time.

A name is looked-up in $\mathcal{O}(t_{\text{look}})$ time for an upper node. Since the number of upper nodes is at most $n/2^\eta$, maintaining the names of the upper nodes takes $\mathcal{O}(t_{\text{look}} n/2^\eta)$ time. This time is subsumed by the lookup time for the η-nodes.

Surnames. Augmenting the (remaining) nodes of the η-truncated HSP tree with surnames cannot be done as simple as in the standard HSP tree construction, since a repetitive node can have a surname equal to the name of a lower node (remember that lower nodes are generated only temporarily, and hence are not maintained in the reverse dictionary). To maintain the surnames pointing to lower nodes, we need to save the names of certain lower nodes in a supplementary reverse dictionary $\mathfrak{D}'$ of $\mathfrak{D}$. This is only necessary when one of the remaining nodes (i.e., the upper nodes and the η-nodes) in the η-truncated HSP tree has a surname that is the name of a lower node. If such a remaining node v is an upper node having a surname equal to the name of a lower node, the η-nodes in the subtree rooted at v have also the same surname. Hence, the number of entries in $\mathfrak{D}'$ is upper bounded by the number of η-nodes. The dictionary $\mathfrak{D}'$ is filled with the surnames of the children of all η-nodes, whose number is at most $3n/2^\eta$. Filling or querying $\mathfrak{D}'$ takes the same time as maintaining the η-nodes. □

Similar to the standard HSP trees, we can conduct LCE queries on two η-truncated HSP trees in the following way:

Lemma 4.32. Let X and Y be two strings, each of length at most n. Given that $\mathsf{tHT}_\eta(X)$ and $\mathsf{tHT}_\eta(Y)$ are built with the *same* dictionary, and given two text positions i_X and i_Y with $1 \leq i_X \leq |X|$ and $1 \leq i_Y \leq |Y|$, we can compute $\mathsf{lcp}(X[i_X\,..], Y[i_Y\,..])$ in $\mathcal{O}(\lg^* n(\lg(n/2^\eta) + 3^\eta/\log_\sigma n))$ time using $\mathcal{O}(\lg(n/2^\eta))$ words of working space.

Proof. Lemma 4.25 gives the time bounds for computing the LCP with two HSP trees. The lemma describes an LCE algorithm that uses the surnames to compare the generated substring of two nodes. By doing so, it accelerates the search for the first pair of mismatching characters in $X[i_X\,..]$ and $Y[i_Y\,..]$. To find this mismatching pair, it examines the subtrees of the two nodes if both nodes mismatch. Since we cannot access a child of an η-node in our η-truncated HSP trees without rebuilding its subtree (as we do not store the lower nodes in $\mathfrak{D}$), we treat the η-nodes as the leaves of the tree. This means that we compare two η-nodes (given their surnames are different) with a naïve

comparison of their generated substrings in $\mathcal{O}(3^\eta/\log_\sigma n)$ time, remembering that the length of the generated substring of an η-node is at most 3^η. For the upper nodes, the algorithm works identically to the original version such that it takes $\mathcal{O}(\lg^* n(\lg(\ell/2^\eta))$ time for traversing those. □

Applying the idea of Cor. 4.26 to Lemma 4.32 gives the following corollary:

Corollary 4.33. Let X and Y be two strings with $|X|, |Y| \leq n$. Given that $\mathsf{tHT}_\eta(X)$ and $\mathsf{tHT}_\eta(Y)$ are built with the *same* dictionary, we can augment both trees with a data structures such that given two text positions $1 \leq i_X \leq |X|, 1 \leq i_Y \leq |Y|$, we can compute $\ell \coloneqq \mathsf{lcp}(X[i_X\,..], Y[i_Y\,..])$ in $\mathcal{O}(\lg^* n(\lg(\ell/2^\eta) + 3^\eta/\log_\sigma n))$ time using $\mathcal{O}(\lg(n/2^\eta))$ words of working space. The additional data structures can be constructed in $\mathcal{O}(n)$ time with $\mathcal{O}(n/\lg n)$ words of space. Their space bounds are within the space bounds of the HSP trees.

Proof. To support accessing the parent of a node in constant time, we construct a pointer based tree structure of the truncated tree during its construction. Since $\mathsf{tHT}_\eta(Y)$ contains at most $n/2^\eta$ nodes, the pointer based tree structure takes $\mathcal{O}(n/2^\eta)$ words.

Given that $\eta \leq \lg\lg n$, we augment the tree structure with a bit vector to jump from a text position to an η-node like in Cor. 4.26: We create a bit vector of length n marking the borders of the generated substrings of the η-nodes such that a rank-support on this bit vector allows us to jump from a position $Y[i]$ to the η-node $\langle Y\rangle_\eta[j]$ with $1+\sum_{k=1}^{j-1} \mathsf{string}(\langle Y\rangle_\eta[k]) \leq i \leq \sum_{k=1}^{j} \mathsf{string}(\langle Y\rangle_\eta[k])$ in constant time. The bit vector with its rank-support takes $\mathcal{O}(n/\lg n)$ words, which is too much to obtain the space bounds of $\mathcal{O}(n/2^\eta)$ words when $\eta = \Omega(\lg\lg n)$.

Instead, we compute a sorted list of pairs if $\eta \geq \log_3(\lg^2 n)$. During the construction of a truncated tree, we collect pairs of constructed η-nodes and their starting positions in a list. This list is automatically sorted by the starting positions as we construct the tree from left to right. The list takes $\mathcal{O}(n/2^\eta)$ words, and we can find the η-node whose generated substring covers a given position in $\mathcal{O}(\lg(n/2^\eta)) = \mathcal{O}(\lg n)$ time by binary searching the starting positions. This time is bounded by the time $\mathcal{O}(\lg^* n\, 3^\eta/\log_\sigma n)$ for scanning the generated substrings of all η-nodes during an LCE query, which is $\mathcal{O}(\lg^* n \lg n \lg\sigma)$ time when $\eta \geq \log_3(\lg^2 n)$.

It is left to consider the case that $\lg\lg n < \eta < \log_3 \lg^2 n$. Let k be the number of η-nodes such that $n/3^\eta \leq k \leq n/2^\eta$. We build the above bit vector in the representation of Pagh [205]. In this representation, the rank-support answers rank queries in constant time. The bit vector together with its rank-support takes $\mathcal{O}(k\lg(n/k) + k^2/n + k(\lg\lg k)^2/\lg k) = \mathcal{O}(k\eta)$ bits (which are $\mathcal{O}(n/2^\eta)$ words) when $k = n/\lg^c n$ for a constant $c > 0$ [210, Thm. 4(b)]. The constant c exists, because $n/\lg^2 n < n/3^\eta \leq k \leq n/2^\eta < n/\lg n$. However, the construction needs $\mathcal{O}(n/\lg n)$ words of space. □

With $\tau \coloneqq 2^\eta$ we obtain the claim of Thm. 4.3.

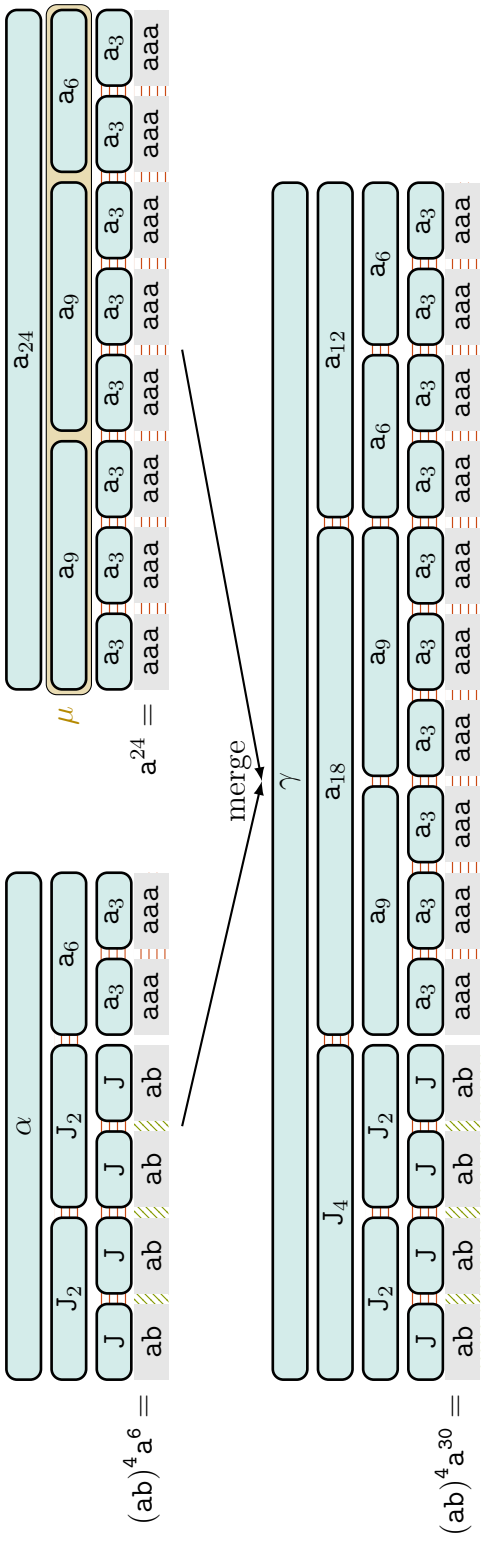


Fig. 4.32: Merging $\mathsf{HT}((\mathsf{ab})^4\mathsf{a}^6)$ with $\mathsf{HT}(\mathsf{a}^{24})$ (both at the top) to $\mathsf{HT}((\mathsf{ab})^4\mathsf{a}^{30})$ (bottom tree). Reparsing the repeating meta-block μ on height one of the right tree is done by recomputing μ's fragile nodes.

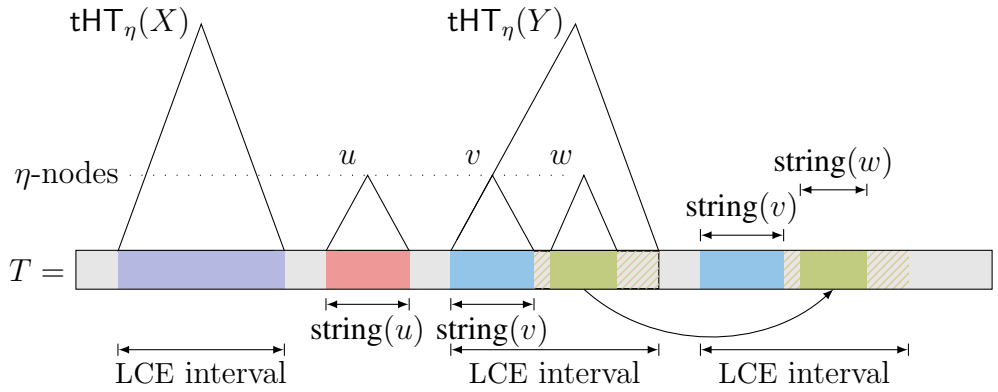


Fig. 4.33: Problem with generated substrings when merging $\mathsf{tHT}_\eta(X)$ and $\mathsf{tHT}_\eta(Y)$. Assume that we want to merge $\mathsf{tHT}_\eta(X)$ and $\mathsf{tHT}_\eta(Y)$, and thus compute the η-nodes (like u) between both trees. On the one hand, we cannot easily find a surrogate substring for the generated substring of a non-surrounded η-node like v or of a newly created η-node like u. Although there is a second occurrence of $\mathsf{string}(v)$ to the right, $\mathsf{string}(v)$ can be extended or shortened when prepending characters (e.g., suppose that $\mathsf{string}(v) = \mathtt{a}^k$, and that there is an $\mathtt{a}$ to the left of the left occurrence of $\mathsf{string}(v)$, but not to the left of the right occurrence). Hence, it is a problem to overwrite $\mathsf{string}(u)$ or the left occurrence of $\mathsf{string}(v)$. On the other hand, we can find suitable surrogate substrings for the generated substrings of the η-nodes like for w that are not near the borders of an LCE interval.

Remark 4.34. In the following, we stick to the result obtained in Lemma 4.32 instead of Cor. 4.33. Although Lemma 4.32 has a slower running time for LCPs that are short, the additional rank-support of Cor. 4.33 makes it difficult to achieve our aimed running time for merging two trees (and therefore would restrain us from achieving our final goal stated in Thm. 4.1). To merge two trees, where each tree is augmented with the bit vector and its rank-support, the task would be to build a rank-support for the concatenation of the bit vectors (preferably in logarithmic time). Unfortunately, we are not aware of a rank-support that is efficiently mergeable (a naïve solution is to build the rank-support of the large bit vector from scratch in linear time).

4.5.2 Sparse Suffix Sorting with Truncated HSP Trees

To use the η-truncated HSP trees as dynLCEs in the situation where they are stored *in text space*, we need an adapted merge operation. Like with HSP trees, merging two η-truncated HSP trees involves a reparsing of the nodes at the facing borders (cf. Fig. 4.32). However, the reparsing of the η-nodes on those borders is especially problematic, as can be seen in Fig. 4.33: Suppose that we rename an η-node v from $\mathtt{N}_2$ to $\mathtt{N}_3$ with $|\mathsf{string}(\mathtt{N}_2)| < |\mathsf{string}(\mathtt{N}_3)|$. If the

name $\mathtt{N}_3$ is not yet maintained in the dictionary, we have to create $\mathtt{N}_3$, i.e., a pointer to a substring X of the text with $X = \mathsf{string}(\mathtt{N}_3)$. The critical part is to find X in the not-yet overwritten parts of the text: Although we can create a suitably long string containing X by concatenating the generated substrings of v's preceding and succeeding siblings, these η-nodes may point to text intervals that are not consecutive. Since the name of an η-node is the representation of a *single* substring, we would have to search X in the *entire* remaining text. In the case that v is surrounded, Lemma 4.22 shows that X is a prefix of the generated substring of a sibling η-node (unlike in Fig. 4.47, where the generated substring of the ESP node with name U cannot be easily determined). With this insight, we finally show an approach that proves Thm. 4.1. For that, it remains to implement Rule 3 and Rule 4 from Sect. 4.4.1 in the context that we maintain η-truncated HSP trees *in text space*: We explain

Goal 1: how the parameter η has to be chosen such that $\mathsf{tHT}_\eta(Y)$ fits into $|Y| \lg \sigma$ bits (needed when creating new trees in Rule 3), and

Goal 2: how to merge two η-truncated HSP trees without the need of extra working space (needed in Rule 4).

4.5.2.1 Storing Truncated HSP Trees in Text Space

Our first goal is to store $\mathsf{tHT}_\eta(T[\mathcal{I}])$ in a text interval $\mathcal{I}$. Since $\mathsf{tHT}_\eta(T[\mathcal{I}])$ can contain nodes with $|\mathcal{I}|/2^\eta$ distinct names, it requires $\mathcal{O}(|\mathcal{I}|/2^\eta)$ words, i.e., $\mathcal{O}(|\mathcal{I}| \lg n/2^\eta)$ bits of space that might not fit in the $|\mathcal{I}| \lg \sigma$ bits of $T[\mathcal{I}]$. Declaring a constant α (independent of n and σ, but dependent on the size of a single node), we can solve this space issue by setting

$$\eta \coloneqq \log_3(\alpha \lg^2 n / \lg \sigma).$$

Lemma 4.35. The number of nodes of an η-truncated HSP tree on a substring of length ℓ is bounded by $\mathcal{O}(\ell(\lg \sigma)^{0.7}/(\lg n)^{1.2})$ with $\eta = \log_3(\alpha \lg^2 n / \lg \sigma)$.

Proof. To obtain the upper bound on the number of nodes, we first compute a lower bound on the number of bits taken by the generated substring of an η-node, which is already lower bounded by $2^\eta \lg \sigma$ bits. We begin with changing the base of the logarithm from 3 to 2/3, and reformulate $\eta = \log_3(\alpha \lg^2 n/\lg \sigma) = (\log_3 2 - 1)\log_{2/3}(\alpha \lg^2 n/\lg \sigma) = \log_{2/3}(\alpha \lg^2 n/\lg \sigma)^{\log_3 2 - 1}$. This gives

$$\begin{aligned} 2^\eta \lg \sigma &= 3^\eta (2/3)^\eta \lg \sigma \\ &= \alpha(\alpha \lg^2 n/\lg \sigma)^{\log_3 2 - 1} \lg^2 n \\ &= (\alpha^{\log_3 2})(\lg n)^{2\log_3 2}(\lg \sigma)^{1-\log_3 2}. \end{aligned}$$

With the estimate $0.6 < \log_3 2 < 0.7$ we simplify this to

$$(\alpha^{\log_3 2})(\lg n)^{2\log_3 2}(\lg \sigma)^{1-\log_3 2} > \alpha^{0.6}(\lg n)^{1.2}(\lg \sigma)^{0.3}.$$

Hence, the generated substring of an η-node takes at least $2^\eta \lg \sigma \geq \alpha^{0.6}(\lg n)^{1.2}(\lg \sigma)^{0.3}$ bits.

Finally, the number of nodes is bounded by

$$\ell/2^\eta \leq \ell \lg \sigma/(\alpha^{0.6}(\lg n)^{1.2}(\lg \sigma)^{0.3}) = \ell(\lg \sigma)^{0.7}/(\alpha^{0.6}(\lg n)^{1.2}). \qquad \square$$

Hence, an η-node with $\eta = \log_3(\alpha \lg^2 n/\lg \sigma)$ generates a substring containing at most $3^\eta = \alpha \lg^2 n/\lg \sigma$ characters.

Plugging this value of η in Lemma 4.31 and Lemma 4.32 yields two corollaries for the η-truncated HSP trees:

Corollary 4.36. We can compute an η-truncated HSP tree on a substring of length ℓ in $\mathcal{O}(\ell \lg^* n + t_{\text{look}}\ell/2^\eta + \ell \lg \lg n)$ time. The tree takes $\mathcal{O}(\ell/2^\eta)$ words of space. We need a working space of $\mathcal{O}(\lg^2 n \lg^* n/\lg \sigma)$ characters.

Proof. The tree has at most $\ell/2^\eta$ nodes, and thus takes $\mathcal{O}(\ell/2^\eta)$ words of space. According to Lemma 4.31, constructing an η-node uses $\mathcal{O}(3^\eta \lg^* n) = \mathcal{O}(\lg^2 n \lg^* n/\lg \sigma)$ characters as working space. $\square$

Corollary 4.37. An LCE query on two η-truncated HSP trees can be answered in $\mathcal{O}(\lg^* n \lg n)$ time.

Proof. LCE queries are answered as in Lemma 4.32, where the time bound depends on η. Since an η-node generates a substring of at most $3^\eta = \alpha \lg^2 n/\lg \sigma$ characters, we can compare the generated substrings of two η-nodes in $\mathcal{O}(\alpha \lg n)$ time. Overall, we compare $\mathcal{O}(\lg^* n)$ η-nodes, such that these additional costs are bounded by $\mathcal{O}(\lg^* n \lg n)$ time overall, and do not slow down the running time $\mathcal{O}(\lg^* n \lg(n/2^\eta) + \lg^* n \lg n) = \mathcal{O}(\lg^* n \lg n)$. $\square$

4.5.2.2 Merging of Truncated HSP Trees

Our second and final goal is to adapt the merging used in the sparse suffix sorting algorithm (Sect. 4.4.1). Suppose that our algorithm finds two intervals $[i\,..\,i+\ell-1]$ and $[j\,..\,j+\ell-1]$ with $T[i\,..\,i+\ell-1] = T[j\,..\,j+\ell-1]$. Ideally, we want to construct $\mathsf{tHT}_\eta(T[i\,..\,i+\ell-1])$ in the text space $[j\,..\,j+\ell-1]$, leaving $T[i\,..\,i+\ell-1]$ untouched so that parts of this substring can be referenced by the η-nodes. Unfortunately, Rules 1 to 4 cannot be applied directly due to our working space limitation. Since we additionally use the text space as working space, we have to be careful about what to overwrite. In particular, we focus on how to

(a) partition the LCE intervals such that the generated substrings of the fragile non-surrounded η-nodes are protected from becoming overwritten,

(b) keep enough working space in text space available for merging two trees,

(c) construct $\mathsf{tHT}_\eta(T[i\,..\,i+\ell-1])$ in the text space $[j\,..\,j+\ell-1]$ when the intervals $[i\,..\,i+\ell-1]$ and $[j\,..\,j+\ell-1]$ overlap, and how to

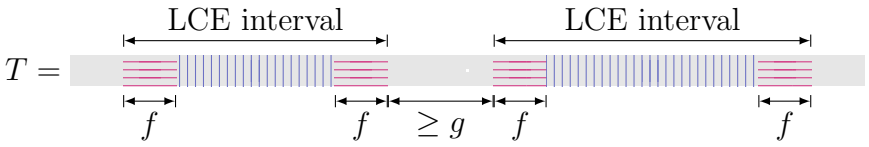


Fig. 4.34: Division of LCE intervals in protected and recyclable parts. The protected and the recyclable parts are depicted with horizontal magenta lines (▤) and vertical violet lines (▥), respectively.

(d) bridge the gap $T[\mathsf{e}(\mathcal{I})+1\,..\,\mathsf{b}(\mathcal{J})-1]$ when merging $\mathsf{tHT}_\eta(T[\mathcal{I}])$ and $\mathsf{tHT}_\eta(T[\mathcal{J}])$ to $\mathsf{tHT}_\eta(T[\mathsf{b}(\mathcal{I})\,..\,\mathsf{e}(\mathcal{J})])$ for two intervals $\mathcal{I}$ and $\mathcal{J}$ with $\mathsf{b}(\mathcal{I}) < \mathsf{b}(\mathcal{J})$ and $|[\mathsf{e}(\mathcal{I})+1\,..\,\mathsf{b}(\mathcal{J})-1]| < g$, as performed in Rule 4.

(a) Partitioning of LCE intervals. To merge two η-truncated HSP trees, we have to take special care of those η-nodes that are fragile, because their names can change due to a merge. If the parsing changes the name of an η-node v, we first check whether v's new name is already present in the dictionary. If it is not, we have to create v's new name consisting of a text position i and a length ℓ such that $T[i\,..\,i+\ell-1] = \mathsf{string}(v)$. The new name of a fragile *surrounded* η-node v can be created easily: According to Lemma 4.22, the generated substring of v is always a prefix of the generated substring of an already existing η-node w, which is found in the reverse dictionary of the η-nodes. Hence, we can create a new name of v with $\mathsf{string}(w)$.

Unfortunately, the same approach does not work with the non-surrounded η-nodes, because those nodes have generated substrings that are found at the borders of $T[j\,..\,j+\ell-1]$ (remember node v of Fig. 4.33). If the characters around the borders are left untouched (meaning that we prohibit overwriting these characters), they can be used for creating the names of the fragile non-surrounded η-nodes during a reparsing. To prevent overwriting these characters, we mark both borders of the interval $[j\,..\,j+\ell-1]$ as protected. Conceptually, we partition an LCE interval into (1) *recyclable* and (2) *protected* intervals (see Fig. 4.34); we free the text of a recyclable interval for overwriting, while prohibiting write access on a protected interval. The recyclable intervals are managed in a dynamic, global list. We comply with the following property:

Property 6: $f := \left\lceil 2\alpha \lg^2 n \Delta_{\mathrm{L}} / \lg \sigma \right\rceil = \Theta(g)$ text positions of the left and right ends of each LCE interval are *protected*.

This property solves the problem for the non-surrounded nodes, because a non-surrounded η-node has a generated substring that is found in $T[j\,..\,j+f-1]$ or $T[j+\ell-1-f\,..\,j+\ell-1]$.

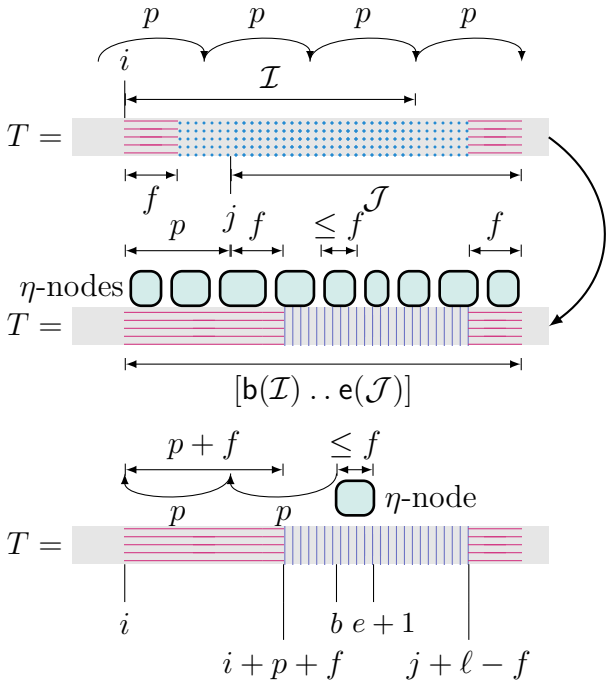


Fig. 4.35: *Top*: Overlapping LCE intervals $\mathcal{I} = [i \,..\, i + \ell - 1]$ and $\mathcal{J} = [j \,..\, j + \ell - 1]$. *Middle*: Partitioning $\mathcal{I}$ and $\mathcal{J}$ as described in (c). *Bottom*: Finding the generated substring $T[b \,..\, e]$ of an η-node in a protected interval. Given that p is a period of $T[\mathcal{I} \cup \mathcal{J}]$, it is sufficient to make $f + p$ characters on the left protected to find the generated substring of all η-nodes of $\mathsf{tHT}_\eta(T[i \,..\, j + \ell - 1])$ in $T[i \,..\, i + p + f - 1]$. The protected and the recyclable parts of the LCE intervals are depicted with horizontal magenta lines (▤) and vertical violet lines (▥), respectively. Parts that have not yet been declared as protected or recyclable are dotted (⁝).

(b) Reserving text space. We can store the upper part of the η-truncated HSP tree in a recyclable interval, because it needs $\ell/2^\eta \lg n \le \ell\alpha^{0.6}(\lg\sigma)^{0.7}/(\lg n)^{0.2} = o(\ell \lg \sigma)$ bits. Since f depends on α and g, we can choose g (the minimum length of a substring on which an η-truncated HSP tree is built) and α (relative to the number of words taken by a single η-truncated HSP tree node) appropriately to always leave $f \lg\sigma/\lg n = \mathcal{O}(\lg^* n \lg n)$ words on a recyclable interval untouched, sufficiently large for the working space needed by Cor. 4.36. Therefore, we precompute α and g based on the input text T, and set both as *global* constants dependent on σ and n. Since the same amount of free space is needed during a subsequent merging when reparsing an η-node, we add the following property:

> Property 7: Each LCE interval has $f \lg\sigma/\lg n$ words of free space left on a recyclable interval.

(c) Interval overlapping. In our algorithm for sparse suffix sorting, a special problem emerges when two computed LCE intervals overlap. For instance, this

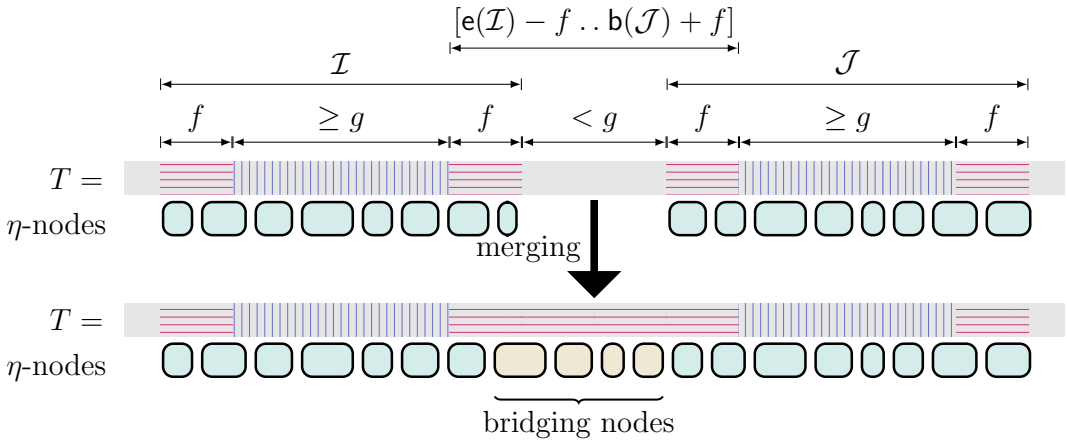


Fig. 4.36: Merging $\mathsf{tHT}_\eta(T[\mathcal{I}])$ and $\mathsf{tHT}_\eta(T[\mathcal{J}])$ with $\mathsf{b}(\mathcal{J}) - g \leq \mathsf{e}(\mathcal{I}) \leq \mathsf{b}(\mathcal{J}) - 1$. The substring $T[\mathsf{e}(\mathcal{I}) - f \,..\, \mathsf{b}(\mathcal{J}) + f]$ is marked protected for the sake of the bridging nodes.

can happen when the LCE of a position $i \in \mathcal{P}$ with a position $j \in \mathcal{P}$ overlaps, i.e.,

$$[i \,..\, i + \mathsf{lce}(i, j) - 1] \cap [j \,..\, j + \mathsf{lce}(i, j) - 1] \neq \emptyset.$$

The algorithm would proceed with merging both overlapping LCE intervals to satisfy Property 5. However, the merged LCE interval cannot respect Property 6 and 7 in general (consider that each interval has a length of $3g$, and both intervals overlap with $2g$ characters). In the case of overlapping, we exploit the periodicity caused by the overlap to make an η-truncated HSP tree fit into both intervals (while still assuring that Property 4 and Property 5 hold, and that we can restore the text).

In a more general setting, suppose that the intervals $\mathcal{I} \coloneqq [i \,..\, i + \ell - 1]$ and $\mathcal{J} \coloneqq [j \,..\, j + \ell - 1]$ with $T[\mathcal{I}] = T[\mathcal{J}]$ overlap, without loss of generality $i < j$. Given $\ell > 2g$, our task is to create $\mathsf{tHT}_\eta(T[i \,..\, j + \ell - 1])$ (e.g., needed to comply with Property 4). Since $T[\mathcal{I}] = T[\mathcal{J}]$, the substring $T[i \,..\, j + \ell - 1]$ has a period p with $1 \leq p \leq j - i$, i.e., $T[i \,..\, j + \ell - 1] = X^k Y$, where $|X| = p$ and Y is a (proper) prefix of X, for an integer k with $k \geq 2$ ($k > 1$ since $j \leq i + \ell - 1$, otherwise $i > j$ or $\mathcal{I} \cap \mathcal{J} = \emptyset$). By definition, each substring of $T[i + p \,..\, j + \ell - 1]$ appears also p characters earlier. We treat the substring $T[i \,..\, i + p + f - 1]$ as a reference and therefore mark it protected. Keeping the original characters in $T[i \,..\, i + p + f - 1]$, we can restore the generated substrings of every η-node by an arithmetic progression. This can be seen by two facts: First, the length of the generated substring of an η-node is at most $3^\eta = \alpha \lg^2 n / \lg \sigma \leq f/2$. Second, given an η-node with the generated substring $T[b \,..\, e]$ with $i + p + f \leq e \leq j + \ell - 1$, we find an integer k with $k \geq 0$ such that $T[b \,..\, e] = T[b - p^k \,..\, e - p^k]$ and $[b - p^k \,..\, e - p^k] \subseteq [i \,..\, i + p + f - 1]$

(since $e-b \le f/2$). Hence, we can make the interval $[i+p+f+1\,..\,j+\ell-1-f]$ *recyclable*, which is at least as large as f, since $|\mathcal{I} \cup \mathcal{J}| \ge j-i+2g \ge p+2g$ is at least $p+3f$ for a sufficiently large g. This partitioning into protected and recyclable intervals is illustrated in Fig. 4.35.

For the actual merging operation, we elaborate an approach that respects Properties 6 and 7:

(d) Merging with a gap. We introduce a merge operation that supports the merging of two η-truncated HSP trees whose LCE intervals have a gap of less than g characters. In contrast to Lemma 4.29, we additionally build new η-nodes on the gap between both trees. The η-nodes whose generated substrings intersect with the gap are called *bridging* nodes.

Let $\mathsf{tHT}_\eta(T[\mathcal{I}])$ and $\mathsf{tHT}_\eta(T[\mathcal{J}])$ be built on two LCE intervals $\mathcal{I}$ and $\mathcal{J}$ with $1 \le \mathsf{b}(\mathcal{J}) - \mathsf{e}(\mathcal{I}) \le g$. Our task is to compute the merged tree $\mathsf{tHT}_\eta(T[\mathsf{b}(\mathcal{I})\,..\,\mathsf{e}(\mathcal{J})])$. We do that by (a) reprocessing $\mathcal{O}(\Delta_\mathrm{L}+\Delta_\mathrm{R})$ nodes at every height of both trees (according to Lemma 4.29), and (b) building the bridging nodes connecting both trees. Like with the non-surrounded nodes, the generated substring of a bridging node can be a unique substring of the text. This means that overwriting $T[\mathsf{e}(\mathcal{I})-f\,..\,\mathsf{b}(\mathcal{J})+f]$ would invalidate the generated substrings of the bridging nodes and of some (formerly) non-surrounded nodes. Therefore, we also mark the interval $[\mathsf{e}(\mathcal{I})-f\,..\,\mathsf{b}(\mathcal{J})+f]$ as protected. By doing so, we can use the characters of $T[\mathsf{e}(\mathcal{I})-f\,..\,\mathsf{b}(\mathcal{J})+f]$ to (a) create the bridging η-nodes, and to (b) reparse the non-surrounded nodes of both trees (Fig. 4.36). The bridging nodes and their ancestors take $o(\lg n \lg^* n)$ words of additional space since building $\mathsf{tHT}_\eta(T[\mathsf{e}(\mathcal{I})+1\,..\,\mathsf{b}(\mathcal{J})-1])$ with $|\mathsf{b}(\mathcal{J})-\mathsf{e}(\mathcal{I})| = \mathcal{O}(g)$ takes $(g/2^\eta)\lg n = o(g\lg\sigma) = o(\lg^* n \lg^2 n)$ bits (or $o(\lg^* n \lg n)$ words) of space. By choosing g and α sufficiently large, we can store the bridging nodes in a recyclable interval while maintaining Property 7 for the merged LCE interval. Finally, the time bound for this merging strategy is given in the following corollary:

Corollary 4.38. Given two LCE intervals $\mathcal{I}$ and $\mathcal{J}$ with $\mathsf{b}(\mathcal{I}) \le \mathsf{b}(\mathcal{J}) \le \mathsf{e}(\mathcal{I})+g$ and their respective η-truncated HSP trees, we can build $\mathsf{tHT}_\eta(T[\mathsf{b}(\mathcal{I})\,..\,\mathsf{e}(\mathcal{J})])$ in $\mathcal{O}(g\lg^* n + t_\mathrm{look} g/2^\eta + g\eta/\log_\sigma n + t_\mathrm{look}\lg^* n \lg n)$ time.

Proof. We adapt the merging of two HSP trees (Lemma 4.29) for the η-truncated HSP trees. The difference to Lemma 4.29 is that we reparse an η-node by rebuilding its local surrounding consisting of $\mathcal{O}((\Delta_\mathrm{L}+\Delta_\mathrm{R})3^\eta)$ nodes that take $\alpha(\Delta_\mathrm{L}+\Delta_\mathrm{R})\lg^2 n/\lg\sigma \le f$ words for a sufficiently large α. According to Property 7, there are at least f words of space left in a recyclable interval to recompute an η-node, and to create the bridging nodes in the fashion of Cor. 4.36. Both creating and recomputing takes overall $\mathcal{O}(g\lg^* n + t_\mathrm{look} g/2^\eta + g\eta/\log_\sigma n)$ time. □

There is one problem left before we can prove the main result of this chapter: The sparse suffix sorting algorithm of Sect. 4.4.1 creates LCE intervals on

substrings smaller than g between two LCE intervals temporarily when applying Rule 3. We cannot afford to build such tiny η-truncated HSP trees, since they cannot respect Property 6 and Property 7. Due to Rule 4, we eventually merge a temporarily created dynLCE with a dynLCE on a long LCE interval. Instead of temporarily creating an η-truncated HSP tree covering less than g characters, we apply the new merge operation of Cor. 4.38 directly, merging two trees that have a gap of less than g characters. With this and the other properties stated above, we come to the final proof:

Proof of Thm. 4.1. The analysis is split into suffix comparison, tree generation and tree merging:

- Suffix comparisons are done as in Cor. 4.28. LCE queries on η-truncated HSP trees and HSP trees are conducted in the same time bounds (compare Lemma 4.25 with Cor. 4.37).
- All positions considered for creating the η-truncated HSP trees belong to $\mathcal{C}$. Constructing the η-truncated HSP trees costs $\mathcal{O}(|\mathcal{C}| \lg^* n + t_{\text{look}} |\mathcal{C}| / 2^\eta + |\mathcal{C}| \lg \lg n)$ overall time, due to Cor. 4.36.
- Merging in the fashion of Cor. 4.38 does not affect the overall time: Since a merge of two trees introduces less than g new text positions to an LCE interval, we conclude with the same analysis as in Thm. 4.30 that the time for merging is upper bounded by the construction time.

Plugging the times for suffix comparisons, tree construction and merging in Cor. 4.28 yields the overall time $\mathcal{O}(t_{\text{C}}(|\mathcal{C}|)) =$

$$\begin{aligned}
&= \mathcal{O}(|\mathcal{C}| \lg^* n + t_{\text{look}} |\mathcal{C}| / 2^\eta + |\mathcal{C}| \lg \lg n) \\
&= \mathcal{O}\Big(|\mathcal{C}| \left(t_{\text{look}} (\lg \sigma)^{0.7} / (\lg n)^{1.2} + \lg \lg n\right)\Big) \\
&= \mathcal{O}\Big(|\mathcal{C}| \left(\sqrt{\lg \sigma} + \lg \lg n\right)\Big)
\end{aligned}$$

because $t_{\text{look}} = \mathcal{O}(\lg n)$. The time for searching and sorting the suffixes is $\mathcal{O}(t_{\text{Q}}(|\mathcal{C}|) m \lg m) = \mathcal{O}(m \lg m \lg^* n \lg n)$. The auxiliary data structures used are $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$, the search tree $\mathcal{L}$ for the LCE intervals, and the list of recyclable intervals, each taking $\mathcal{O}(m)$ words of space. □

4.6 Alternative to the Suffix AVL Tree

Instead of using the suffix AVL tree of Irving and Love [134], we can devise an alternative data structure for computing the sparse suffix sorting: A balanced binary search tree (e.g., an AVL or red-black tree) with suffixes as keys and the lexicographic order as sorting criterion. Each node of the tree is augmented with four LCP values. We call this data structure a binary search prefix tree (BSPT),

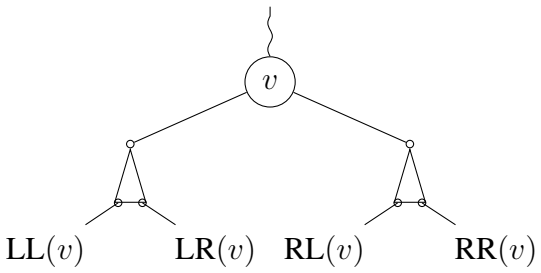


Fig. 4.37: Node $v \in \mathcal{B}(T, \mathcal{P})$ with two children. Each child has a subtree, which is symbolized by a triangle. The rightmost and leftmost leaves of these two subtrees define the nodes $\mathrm{LL}(v)$, $\mathrm{LR}(v)$, $\mathrm{RL}(v)$, and $\mathrm{RR}(v)$.

and write $\mathcal{B}(T, \mathcal{P})$ for the BSPT containing suffixes beginning at the text positions of a set $\mathcal{P}$. We name its nodes after the starting positions of their corresponding suffixes, i.e., a node $v \in \mathcal{B}(T, \mathcal{P})$ corresponds to the suffix $T[v\,..]$ (like the suffix tree leaves storing suffix numbers in Sect. 3.3.2). A node v stores the LCP values (see also Fig. 4.37)

- $\mathsf{cp}_{\mathtt{LL}}(v) \coloneqq \mathsf{lce}(v, \mathrm{LL}(v))$,
- $\mathsf{cp}_{\mathtt{LR}}(v) \coloneqq \mathsf{lce}(v, \mathrm{LR}(v))$,
- $\mathsf{cp}_{\mathtt{RL}}(v) \coloneqq \mathsf{lce}(v, \mathrm{RL}(v))$, and
- $\mathsf{cp}_{\mathtt{RR}}(v) \coloneqq \mathsf{lce}(v, \mathrm{RR}(v))$,

where

- $\mathrm{LL}(v)$ (resp. $\mathrm{LR}(v)$) denotes the leftmost (resp. rightmost) node in the subtree rooted at the *left* child of v, and
- $\mathrm{RL}(v)$ (resp. $\mathrm{RR}(v)$) denotes the leftmost (resp. rightmost) node in the subtree rooted at the *right* child of v.

If v does not have a left (resp. right) child, set $\mathrm{LL}(v), \mathrm{LR}(v) \leftarrow v$ (resp. $\mathrm{RL}(v), \mathrm{RR}(v) \leftarrow v$) for convenience. Using a pointer based structure for the tree, $\mathcal{B}(T, \mathcal{P})$ occupies $\mathcal{O}(|\mathcal{P}|)$ words of space. It has the following properties:

- $\mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP}(T, \mathcal{P})$ can be computed by an in-order traversal of the tree. We have $\mathsf{SSA}(T, \mathcal{P})[i] = v$ for a node v with in-order number i. Additionally, $\mathsf{SLCP}(T, \mathcal{P})[i] = \mathsf{cp}_{\mathtt{LR}}(v)$ (resp. $\mathsf{SLCP}(T, \mathcal{P})[i+1] = \mathsf{cp}_{\mathtt{RL}}(v)$) if v has a left (resp. right) child,
- Accessing $\mathsf{SSA}(T, \mathcal{P})$ or $\mathsf{SLCP}(T, \mathcal{P})$ can be supported in $\mathcal{O}(\lg |\mathcal{P}|)$ time, given that we augment each node with its subtree size.

This shows that the BSPT is a more natural choice for representing $\mathsf{SSA}(T, \mathcal{P})$ and $\mathsf{SLCP}(T, \mathcal{P})$, compared to the suffix AVL tree (cf. above of Cor. 4.28). It can also perform pattern matching within the same time bounds as the suffix AVL tree. To show this, we need a small helper lemma:

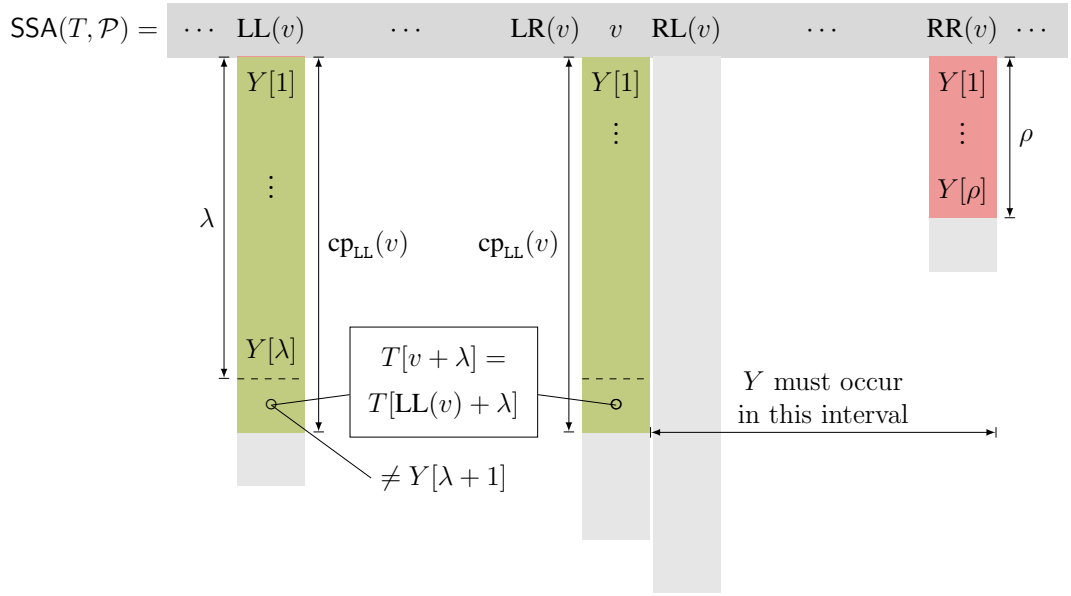


Fig. 4.38: Setting of the proof of Thm. 4.40(a), where $\mathsf{cp}_{\mathtt{LL}}(v) > \lambda$. The vertical bar below the entry $\mathsf{SSA}(T, \mathcal{P})[i]$ is of length $\mathsf{SLCP}(T, \mathcal{P})[i]$. Since $\lambda = \mathsf{lcp}(T[\mathbf{LL}(v)\,..], Y)$, $Y[j] = T[\mathbf{LL}(v) + j - 1]$ for all $j = 1, \ldots, \lambda$, but $T[\mathbf{LL}(v) + \lambda] = T[v + \lambda] \neq Y[\lambda]$.

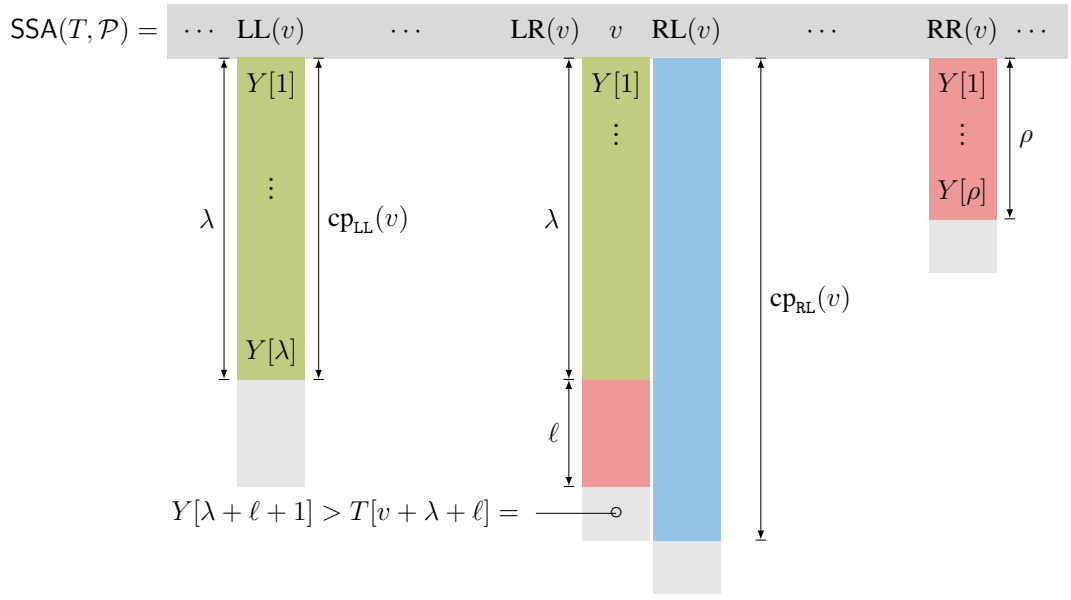


Fig. 4.39: Setting of the proof of Thm. 4.40(b), where $\mathsf{cp}_{\mathtt{LL}}(v) = \lambda$. The figure depicts the particular case that $T[v + \lambda + \ell] < Y[\lambda + \ell + 1]$ and $\mathsf{cp}_{\mathtt{RL}}(v) \geq \lambda + \ell$. All occurrences of Y are found in the right subtree of v.

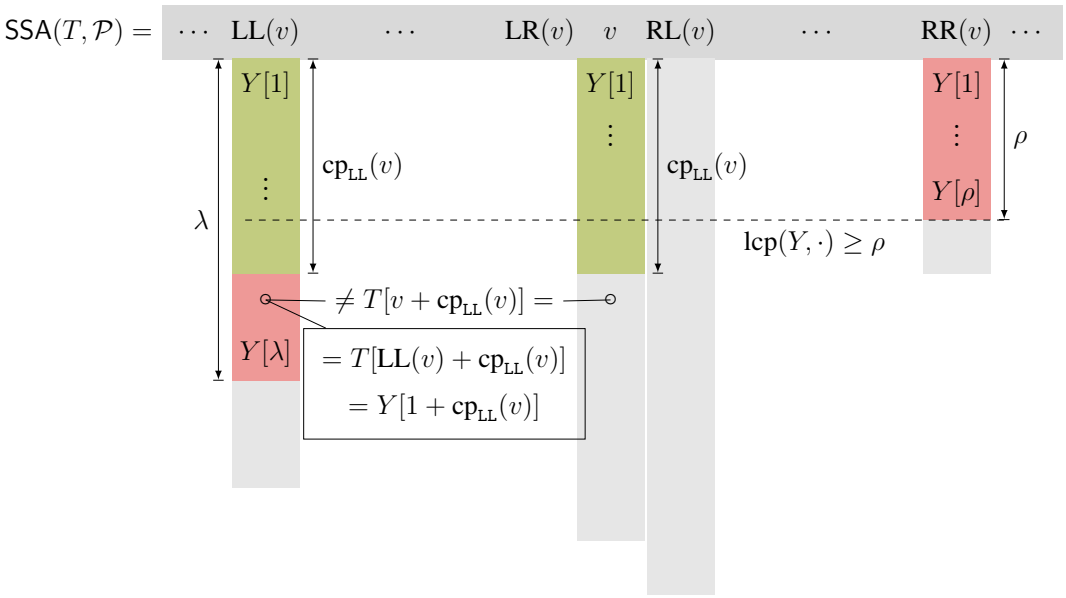


Fig. 4.40: Setting of the proof of Thm. 4.40(c), where $\mathsf{cp}_{\mathrm{LL}}(v) < \lambda$. We have $Y[j] = T[\mathrm{LL}(v) + j - 1] = T[v + j - 1]$ for all $j = 1, \ldots, \mathsf{cp}_{\mathrm{LL}}(v)$, but $Y[1 + \mathsf{cp}_{\mathrm{LL}}(v)] = T[\mathrm{LL}(v) + \mathsf{cp}_{\mathrm{LL}}(v)] \neq T[v + \mathsf{cp}_{\mathrm{LL}}(v)]$. Since $\mathsf{lcp}(Y, T[i\,..]) \geq \rho$ for all $i = \mathsf{cp}_{\mathrm{LL}}(v), \ldots, \mathsf{cp}_{\mathrm{RR}}(v)$, all occurrences of Y are found in the left subtree of v.

Lemma 4.39 ([134, Lemma 1])**.** Given three strings X, Y, Z with the lexicographic order $X \prec Y \prec Z$, we have $\mathsf{lcp}(X, Z) = \min(\mathsf{lcp}(X, Y), \mathsf{lcp}(Y, Z))$.

Next, we show that we can search a pattern Y within the same time bounds as the suffix AVL tree:

Theorem 4.40. Given a text T of length n, and $\mathcal{B}(T, \mathcal{P})$ built on the suffixes of T whose starting positions are in $\mathcal{P}$, we can find $\operatorname{argmax}\{\mathsf{lcp}(Y, T[p\,..]) \mid p \in \mathcal{P}\}$ of a pattern Y in $\mathcal{O}(|Y| / \log_\sigma n + \lg |\mathcal{P}|)$ time.

Proof. Analogously to the pattern matching algorithm with SA and LCP [183, Fig. 3], we perform a binary search by walking down the tree. We maintain two variables $\lambda, \rho \in [1\,..\,n]$ with the invariant that $\lambda = \mathsf{lcp}(T[\mathrm{LL}(v)\,..], Y)$ and $\rho = \mathsf{lcp}(T[\mathrm{RR}(v)\,..], Y)$ on visiting a node v. Starting at the root note, we initialize $\lambda \leftarrow \mathsf{lcp}(T[\mathrm{LL}(\mathsf{root})\,..], Y)$ and $\rho \leftarrow \mathsf{lcp}(T[\mathrm{RR}(\mathsf{root})\,..], Y)$, where root is the root node of $\mathcal{B}(T, \mathcal{P})$. Suppose that we are currently at a node $v \in \mathcal{B}(T, \mathcal{P})$. Further suppose that $\lambda \geq \rho$. Otherwise ($\lambda < \rho$), exchange

- $\mathrm{LL}(v)$, $\mathrm{LR}(v)$, $\mathrm{RL}(v)$, $\mathrm{RR}(v)$, and λ with
- $\mathrm{RR}(v)$, $\mathrm{RL}(v)$, $\mathrm{LR}(v)$, $\mathrm{LL}(v)$, and ρ, respectively.

We follow the case analysis of [183, Fig. 2], whose cases depend on the relationship between $\mathsf{cp}_{\mathrm{LL}}(v)$ and λ:

(a) Case $\mathsf{cp}_{\mathtt{LL}}(v) > \lambda$, see also Fig. 4.38. Then $Y[\lambda+1] > T[v+\lambda] = T[\mathbf{LL}(v)+\lambda]$, and thus $Y \succ T[v\,..]$.

- If $\mathsf{cp}_{\mathtt{RL}}(v) < \lambda$ then Y does not occur in T.
- Otherwise ($\mathsf{cp}_{\mathtt{RL}}(v) \geq \lambda$), $\lambda \leq \mathsf{lcp}(Y[1\,..], T[\mathbf{RL}(v)\,..])$. We set $\lambda \leftarrow \lambda + \mathsf{lcp}(Y[1+\lambda\,..], T[\mathbf{RL}(v)+\lambda\,..])$, and descend to v's right child.

(b) Case $\mathsf{cp}_{\mathtt{LL}}(v) = \lambda$, see also Fig. 4.39. We compute $\ell \coloneqq \mathsf{lcp}(T[v+\lambda\,..], Y[\lambda+1\,..])$. If both strings are equal, we found a match and return. Otherwise, we compare the first pair of mismatching characters $T[v+\lambda+\ell]$ and $Y[\lambda+\ell+1]$.

- If $T[v+\lambda+\ell] < Y[\lambda+\ell+1]$, then the corresponding suffix $T[v\,..]$ of v is (lexicographically) smaller than Y. We abort the search if $\mathsf{cp}_{\mathtt{RL}}(v) < \lambda+\ell$, because then the next lexicographically larger suffix $T[\mathbf{RL}(v)\,..]$ stored in $\mathcal{B}(T,\mathcal{P})$ shares a shorter prefix with Y than $T[v\,..]$ with Y. Under the assumption that $\mathsf{cp}_{\mathtt{RL}}(v) \geq \lambda+\ell$, we set $\lambda \leftarrow \mathsf{lcp}(T[\mathbf{RL}(v)+\lambda+\ell\,..], Y[\lambda+\ell+1\,..])$, and descend to v's right child.
- The case that $T[v+\lambda+\ell] > Y[\lambda+\ell+1]$ is symmetrical: $T[v\,..]$ is larger than Y. We abort the search if $\mathsf{cp}_{\mathtt{LR}}(v) < \lambda+\ell$, because then $\mathsf{lcp}(T[\mathbf{LR}(v)\,..], Y) < \mathsf{lcp}(T[v\,..], Y)$. Under the assumption that $\mathsf{cp}_{\mathtt{LR}}(v) \geq \lambda+\ell$, we set $\rho \leftarrow \mathsf{lcp}(T[\mathbf{LR}(v)+\lambda+\ell\,..], Y[\lambda+\ell+1\,..])$, and descend to v's left child.

(c) Case $\mathsf{cp}_{\mathtt{LL}}(v) < \lambda$, see also Fig. 4.40. We have $\mathsf{cp}_{\mathtt{LL}}(v) \geq \rho$ since $Y[1\,..\,\rho]$ is a common prefix of all suffixes corresponding to the nodes of v's subtree (according to the assumption that $\rho < \lambda$). In particular, $\rho \leq \mathsf{lcp}(Y[1\,..], T[\mathbf{LR}(v)\,..])$. We set $\rho \leftarrow \rho + \mathsf{lcp}(Y[1+\rho\,..], T[\mathbf{LR}(v)+\rho\,..])$, and descend to v's left child.

We never decrease ρ and λ (instead we abort in the case that Y is not a prefix of any suffixes). Since both values are upper bounded by $|Y|$, we compare $\mathcal{O}(|Y|)$ characters in total. In the word-packing model, this gives a running time of $\mathcal{O}(|Y| / \log_\sigma n)$ for matching the characters of Y. Since the tree is balanced, we access $\mathcal{O}(\lg|\mathcal{P}|)$ nodes. In total, the pattern matching takes $\mathcal{O}(|Y| / \log_\sigma n + \lg|\mathcal{P}|)$ time. □

It remains to show how to insert new suffixes into $\mathcal{B}(T,\mathcal{P})$ efficiently. Let $1 \leq p \leq |T|, p \notin \mathcal{P}$ be a text position that we want to add to $\mathcal{B}(T,\mathcal{P})$. We locate the insertion point in $\mathcal{B}(T,\mathcal{P})$, insert a new leaf, update all invalidated LCP values, and re-balance the tree, if necessary. With the aid of Thm. 4.40, we can update the LCP values efficiently:

Insertion Suppose that our goal is to insert v as the left child of a node u_1 (inserting the right child is analogous by symmetry), see also Fig. 4.41.

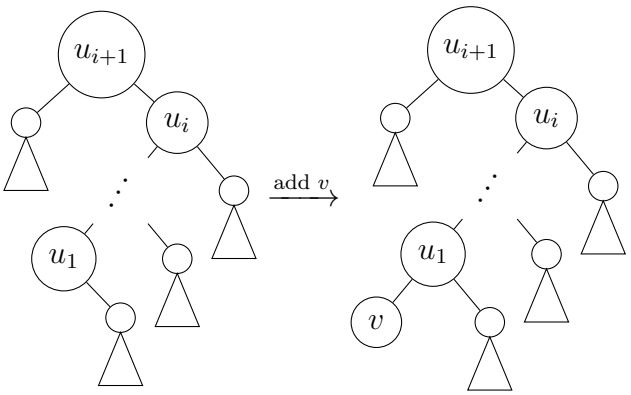


Fig. 4.41: Creating the left child v of a node u_1 of a BSPT. The node u_{i+1} is the lowest node on the path from u_1 to the root that has an ancestor of u_1 as its right child.

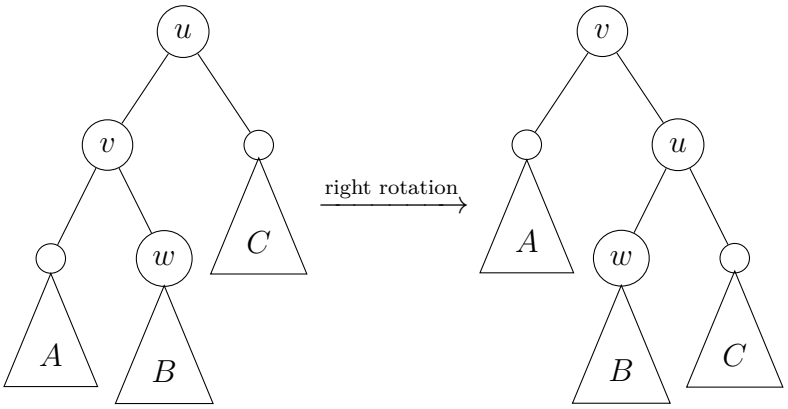


Fig. 4.42: Right-rotating a BSPT. The BSPT is depicted before (*left*) and after (*right*) right-rotating the triple of nodes (u, v, w).

First we update the stored LCP values in $\mathcal{B}(T, \mathcal{P})$, and subsequently perform rotations (if necessary). Let $u_1, u_2, \ldots, u_i$ be the maximal sequence of the ancestors of v such that u_j is left child of u_{j+1} for every integer j with $1 \leq j \leq i-1$. According to Lemma 4.39 we update

$$\mathsf{cp}_{\mathtt{LL}}(u_j) \leftarrow \mathsf{lcp}(T[u_j\,..], T[u\,..]) = \min(\mathsf{lcp}(T[u\,..], T[u_1\,..]), \mathsf{cp}_{\mathtt{LL}}(u_j))$$

for every j with $1 \leq j \leq i$, and $\mathsf{cp}_{\mathtt{RL}}(u_{i+1}) \leftarrow \mathsf{lcp}(T[u_{i+1}\,..], T[v\,..])$, where u_{i+1} is the parent of u_i (we omit u_{i+1} if u_i is the root node). In total, we need to compute the two LCP queries $\mathsf{lcp}(T[v\,..], T[u_1\,..])$ and $\mathsf{lcp}(T[u_{i+1}\,..], T[v\,..])$, whose values have already been computed after locating the insertion point u_1 as described in Thm. 4.40.

Rotation Suppose that we need to right-rotate the triple of nodes (u, v, w), where w is the right child of v that is the left child of u. This setting is also depicted in Fig. 4.42. The right rotation makes (a) w the left child of u, and (b) u the right child of v. The operations (a) and (b) invalidate the values $\mathsf{cp}_{\mathtt{LL}}(u)$ and $\mathsf{cp}_{\mathtt{RR}}(v)$, respectively. With an application of Lemma 4.39, we can restore these values by setting

(a) $\mathsf{cp}_{\mathtt{LL}}(u) \leftarrow \min(\mathsf{cp}_{\mathtt{LR}}(u), \mathsf{cp}_{\mathtt{RR}}(w), \mathsf{cp}_{\mathtt{LL}}(w))$ (cf. Fig. 4.43), and

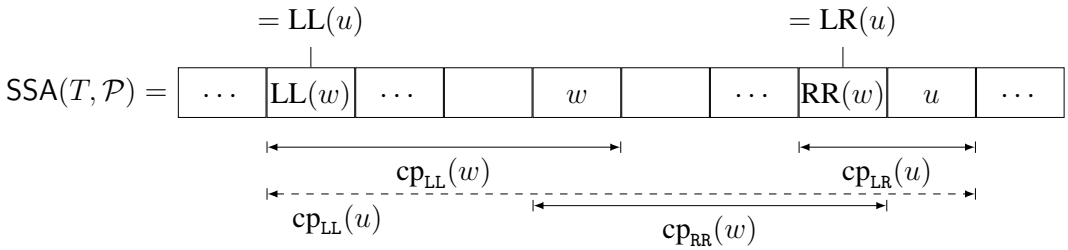


Fig. 4.43: Updating the value $\mathrm{cp}_{\mathtt{LL}}(u)$ after the right rotation of Fig. 4.42.

(b) $\mathrm{cp}_{\mathtt{RR}}(v) \leftarrow \min(\mathrm{cp}_{\mathtt{RR}}(v), \mathrm{cp}_{\mathtt{LR}}(u), \mathrm{cp}_{\mathtt{RR}}(u))$.

By symmetry, left rotations are done analogously.

Lemma 4.41. Updating $\mathcal{B}(T, \mathcal{P})$ to $\mathcal{B}(T, \mathcal{P} \cup \{p\})$ can be done in $\mathcal{O}(\ell / \log_\sigma n + \lg n)$ time, where $p \in [1 \,..\, n]$ and $\ell = |T[p \,..]|$.

Like with the suffix AVL tree, it is possible to generalize BSPT to maintain general strings instead of suffixes. For instance, BSPT could be used for sorting strings online.

4.7 Conclusion

In the first part, we introduced the HSP trees based on the ESP technique as a new data structure that (a) answers LCE queries, and (b) can merge with another HSP tree to form a larger HSP tree. With these properties, HSP trees are an eligible choice for the mergeable LCE data structure needed for the sparse suffix sorting algorithm presented here.

In the second part, we developed a truncated version of the HSP tree with a trade-off parameter determining the height at which to cut off the lower nodes. Setting the trade-off parameter adequately, the truncated HSP tree fits into text-space. As a result of independent interest, we obtained an LCE data structure with a trade-off parameter, like other already known solutions. Although not shown here, an ESP tree can similarly (a) answer LCE queries, (b) be merged, and (c) be truncated. However, answering LCE queries or merging two ESP trees is by a factor of $\mathcal{O}(\lg n)$ slower than when the operations are performed with HSP trees.

We also noted that the maximum number of fragile nodes in an ESP tree of a string of length n can be at least $\Omega(\lg^2 n)$, which invalidates the upper bound of $\mathcal{O}(\lg n \lg^* n)$ on the maximal number of fragile nodes postulated in [55]. This result also invalidates theoretical results that depend on the ESP technique (e.g., for approximating the SEDM [55] or the LZ77 factorization [54], or for building indexes [100, 186, 227, 228]). We could quickly provide a new upper bound of $\mathcal{O}(\lg^2 n \lg^* n)$, but it remains an open problem to refine our bounds. Luckily,

our proposed HSP technique can be used as a substitution for the ESP technique, since HSP trees and ESP trees share the same bounds for construction time and space usage. By switching to the HSP technique, we regain the promised $\mathcal{O}(\lg n \lg^* n)$ number of fragile nodes. It is easy to see that this result also recovers the postulated $\mathcal{O}(\lg n \lg^* n)$ approximation bound on the edit distance matching problem [55, 228]: Given $\mathsf{ET}(T)$ of a string T of length n, it is assumed by Cormode and Muthukrishnan [55, Thm. 7] that changing/deleting a character of T or inserting a character in T changes $\mathcal{O}(\lg^* n \lg n)$ nodes in $\mathsf{ET}(T)$. Although we only provided proofs that pre-/appending characters to T changes $\mathcal{O}(\lg^* n \lg n)$ nodes of $\mathsf{HT}(T)$, it is easy to generalize this result by applying a merge operation: Given that we insert a character $\mathsf{c} \in \Sigma$ between $T[i]$ and $T[i{+}1]$, the trees $\mathsf{HT}(T)$ and $\mathsf{HT}(T[1\,..\,i]\mathsf{c}T[i+1\,..])$ differ in at most $\mathcal{O}(\lg^* n \lg n)$ nodes, since appending c to $\mathsf{HT}(T[1\,..\,i])$ and merging $\mathsf{HT}(T[1\,..\,i]\mathsf{c})$ with $\mathsf{HT}(T[i+1\,..])$ changes $\mathcal{O}(\lg^* n \lg n)$ nodes. The same can be observed when deleting or changing the i-th character.

Our open problems are:

Practical evaluation. In the light of the theoretical improvements of the HSP over the ESP, it is interesting to evaluate how the HSP behaves practically. Especially, we are interested in how well the HSP behaves in the context of grammar compression [19] like the ESP-index [186, 227] on highly repetitive texts, where a more stable behavior of the repetitive nodes could lead to an improved compression ratio.

Suffix sorting with trade-off parameter. From the theoretical point of view, it would be interesting to compute the sparse suffix sorting with a trade-off parameter adjusting working space and construction time of SSA and SLCP.

Sparsity based on suffix selection. In the case that we can impose a restriction on the set of suffixes to sort, Kärkkäinen and Ukkonen [144] presented a sparse suffix sorting algorithm running in optimal $\mathcal{O}(n)$ time while using $\mathcal{O}(m)$ words of space, given that $\mathcal{P}$ is a set of equally spaced text positions. We[9] wonder whether it is also possible to gain a benefit when only every i-th entry of SA is needed, i.e., the order of each i-th lexicographically smallest suffix for an arithmetic progression $i = c, 2c, 3c, \ldots$ with a constant integer $c \geq 2$. Related to this problem is the suffix selection problem, i.e., to find the i-th lexicographically smallest suffix for a given integer i. Interestingly, Franceschini and Muthukrishnan [96] showed that the suffix selection problem can be solved in $\mathcal{O}(n)$ time in the comparison model, whereas suffix sorting is solved in $\Theta(n \lg n)$ time within the same model.

[9] The idea of this problem was worked out together with Moshe Lewenstein.

Mergeable rank-support. Remembering Remark 4.34, we are unaware of whether rank-support data structures can be mergeable. Given two bit vectors B_1 and B_2, both with a rank-support data structure, the task is to compute a rank-support data structure on the concatenation of B_1 and B_2 in sub-linear time in the total lengths of both bit vectors.

Construction space aware compressed bit vectors. Although there are bit vectors with rank-support that can be stored in compressed space (e.g., [205]), there is, to the best of our knowledge, no (compressed) bit vector representation that can be constructed within compressed space or online.

Sparse LZ77 factorization. The LZ77 factorization of Sect. 3.4 could be performed on the sparse suffix tree, i.e., the suffix tree, in which all leaves not corresponding to the m selected suffixes are omitted. Then the factorization can only create referencing factors at the m selected positions. The referencing factors have a referred position within $\mathcal{P}$.

Extracting $\mathsf{SLCP}(T, \mathcal{P})$ from $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$. Each node of $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ represents a suffix starting at a position of $\mathcal{P}$. Besides a starting position, a node v additionally stores $\max\{\mathsf{lcp}(u, v) \mid u \text{ ancestor of } v\}$. The node u is one of the two lowest ancestors having v in either its left or right subtree. In case that such an ancestor u exists (the stored LCP value is greater than zero), v additionally records whether it is in u's left or right subtree (using a flag bit), such that v and this bit uniquely determine the position of u in the tree. Although we can transform $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ to $\mathsf{SSA}(T, \mathcal{P})$ with a simple in-order traversal as shown in Cor. 4.28, it is not obvious whether $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ holds enough information to determine the contents of $\mathsf{SLCP}(T, \mathcal{P})$. A conjecture is that the stored LCP values in $\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$ are actually a permutation of $\mathsf{SLCP}(T, \mathcal{P})$, where missing values can be computed by taking the minimum values of certain ranges.

4.8 Landscape Oriented Figures

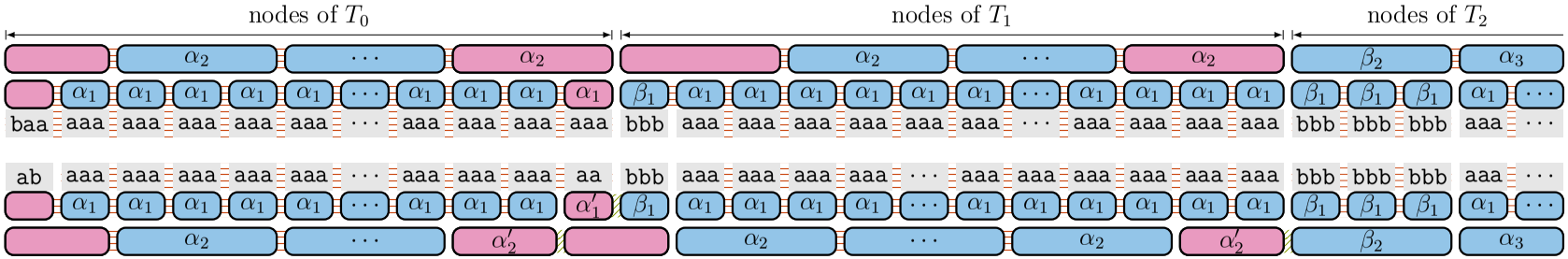


Fig. 4.44: Excerpt of the ESP trees $\mathsf{ET}(Y)$ (*top*) and $\mathsf{ET}(Y')$ (*bottom*, vertically flipped), where $Y = \mathtt{ba}^{3^k-1}\mathtt{b}^3\mathtt{a}^{3^k-3}\mathtt{b}^9\mathtt{a}^{3^k-9}\cdots$ and $Y' = \mathtt{a}Y$ (defined in Thm. 4.13). The two trees differ in the nodes that are highlighted in magenta (■). Note that right of the rightmost α'_2 (bottom tree, rightmost magenta node) is the node β_2, and both nodes form a Type 2 meta-block.

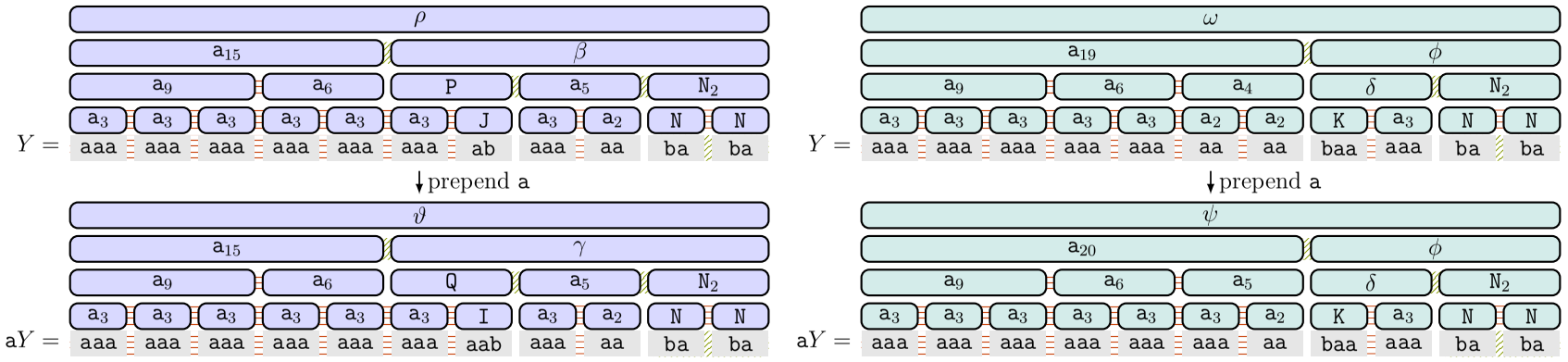


Fig. 4.45: Impact of the tie breaking rule (Rule (M)) on emerging Type M nodes of the HSP trees built on $Y = \mathtt{a}^{19}\mathtt{ba}^{5}(\mathtt{ba})^{2}$. A Type M node (like I, J on the left or K on the right) is created by fusing a single symbol with its sibling meta-block. Remember that Rule (M) prescribes to fuse the symbol with its *succeeding* meta-block. To see why this rule is advantageous, the HSP trees on the *left* (resp. *right*) use the tie breaking rule Rule (M') (resp. Rule (M)) favoring the *preceding* (resp. *succeeding*) meta-block. While on the *right* side only the fragile nodes of the leftmost meta-blocks on each height differ after prepending a (e.g., the unique occurrence of $\mathtt{a}_4$ changes to $\mathtt{a}_5$), the change is more dramatical on the *left* side. Prepending the character a to Y (*bottom left*) changes the names of the nodes with names J and P to I and Q, respectively.

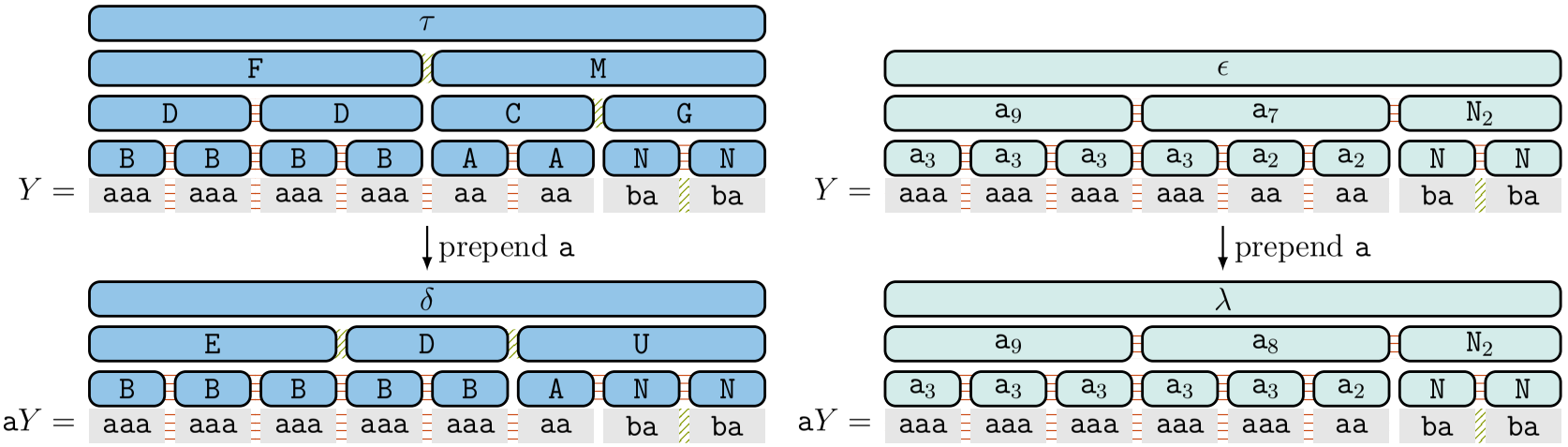


Fig. 4.46: *Top*: $\mathsf{ET}(Y)$ (*left*) and $\mathsf{HT}(Y)$ (*right*) of the string Y defined in Fig. 4.7. *Bottom*: $\mathsf{ET}(\mathtt{a}Y)$ (*left*) and $\mathsf{HT}(\mathtt{a}Y)$ (*right*). Unlike the two ESP trees at the *left*, the two HSP trees at the *right* share the same tree topology.

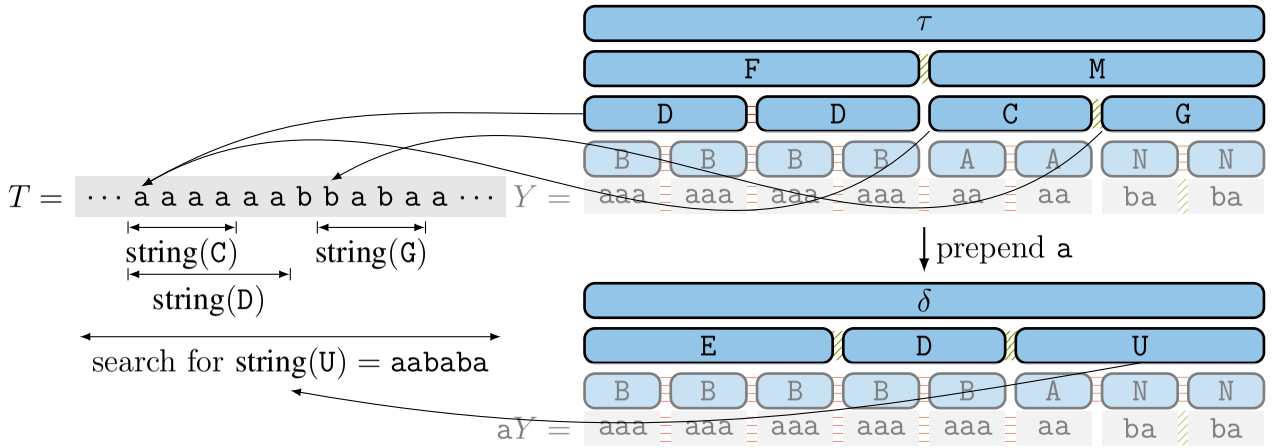


Fig. 4.47: Problem with dynamic updates of ESP trees stored in text space. Suppose that we truncate ESP trees at a certain height. Truncated nodes are grayed out. Each leaf of the truncated trees is assigned a pointer to its generated substring, which is a substring of the text T (*left*). Suppose that we have built $\mathsf{ET}(Y)$ (*top right*) on a substring Y of T (Y defined as in Fig. 4.46), and that the names D, C and G are already present in the dictionary (hence, they have different generated substrings). Further suppose that the space of Y in T has been overwritten. When prepending an a to $\mathsf{ET}(Y)$ to form $\mathsf{ET}(\mathtt{a}Y)$ (*bottom right*), the node G changes to U, for which we need to search its generated substring (assuming that U is not yet stored in the dictionary). The example can be elaborated such that G and U become surrounded nodes (prepend $\mathtt{a}^{9k}$ and append $\mathtt{b}^{9k}$ for a sufficiently large $k \geq 1$).

Chapter

5

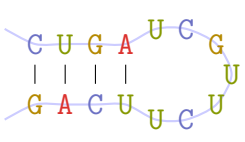
Gapped Regular Structures

The laws of Nature are based upon the existence of a pattern, linking one state of affairs to another; and where there is pattern, there is symmetry.

— John D. Barrow [24]

Gapped regular structures, i.e., gapped repeats and gapped palindromes, are a generalization of squares and palindromes. They were investigated extensively within theoretical computer science [60, 64, 66, 70, 161, 163, 166, 229], motivated by DNA and ribonucleic acid (RNA) structures [120, Sect. 7.11.2], modelling different types of tandem repeats (i.e. repetitions), interspersed repeats (e.g., [139, 222]), as well as inverted repeats (e.g., [234, 240]) or hairpin structures (e.g., [226] and Fig. 5.1). Such structures are important for the analysis of structural and functional information of genetic sequences, as can be seen by the number of software products devoted to finding those (see references in [67, 170, 232]).

Given a text T, a *gapped repeat* is a triple of integers $(i_{\mathrm{L}}, i_{\mathrm{R}}, u)$ with the properties (a) $0 < i_{\mathrm{R}} - i_{\mathrm{L}}$ and (b) $T[i_{\mathrm{L}} \,..\, i_{\mathrm{L}} + u - 1] = T[i_{\mathrm{R}} \,..\, i_{\mathrm{R}} + u - 1]$. A variant of gapped repeats are *gapped palindromes* with the properties (a) $0 \leq i_{\mathrm{R}} - i_{\mathrm{L}}$ and (b) $T[i_{\mathrm{L}} \,..\, i_{\mathrm{L}} + u - 1]$ is equal to the reverse of $T[i_{\mathrm{R}} \,..\, i_{\mathrm{R}} + u - 1]$. In both cases (repeats or palindromes), $T[i_{\mathrm{L}} \,..\, i_{\mathrm{L}} + u - 1]$ and $T[i_{\mathrm{R}} \,..\, i_{\mathrm{R}} + u - 1]$ are called *left* and *right arm*, respectively. Given a real number $\alpha \geq 1$, a gapped repeat or palindrome $(i_{\mathrm{L}}, i_{\mathrm{R}}, u)$ is called *α-gapped* if $i_{\mathrm{R}} - i_{\mathrm{L}} \leq \alpha u$. A gapped repeat and a gapped palindrome of a text of length n are an n-gapped repeat and an n-gapped palindrome, respectively. A gapped repeat is *maximal* if its arms



$\varphi(\mathtt{A}) = \mathtt{U}$
$\varphi(\mathtt{C}) = \mathtt{G}$
$\varphi(\mathtt{G}) = \mathtt{C}$
$\varphi(\mathtt{U}) = \mathtt{A}$

Fig. 5.1: An RNA hairpin structure, reading CUGAUCGUUCUUCAG. Each character represents an RNA base. The vertical bars connect bases according to their Watson-Crick base complement. Given that φ maps bases to their complements, $\varphi(\mathtt{CUGA})$ is equal to GACU, which is the reverse of UCAG. Both endings connected by the vertical bars define the arms of a maximal gapped φ-palindrome, defined in Remark 5.35.

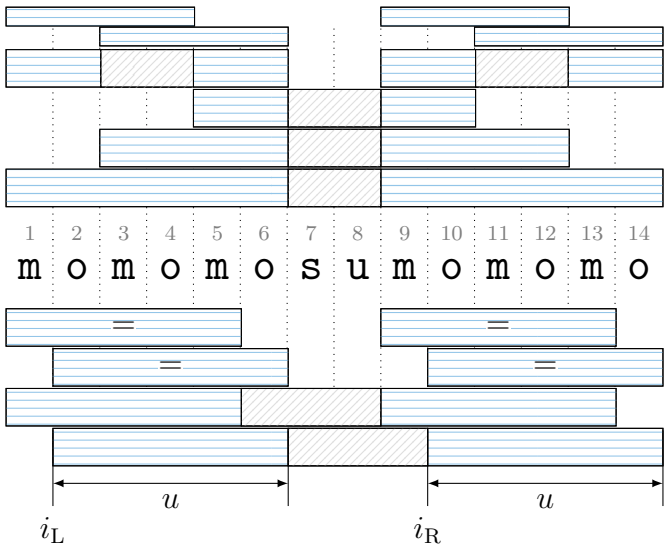


Fig. 5.2: Maximal 2-gapped repeats (*above*) and palindromes (*below*) of the string `momomosumomomo`. The arms are the blocks with the blue horizontal hatching (▤), the gap between two arms belonging to the same gapped repeat or palindrome is diagonally hatched (▨). The gapped palindrome at the *bottom* is described by the triple $(i_L, i_R, u) = (2, 10, 5)$. The gapped repeats at the *top* are repeats with overlapping arms (symbolized by halving their height). The arms of gapped palindromes with label = coincide.

cannot be extended to their left nor to their right sides (to form a larger gapped repeat). Similarly, a gapped palindrome is *maximal* if it can be extended neither inwards (shrinking the gap on both sides by one character) nor outwards. The set of all maximal α-gapped repeats and palindromes of a text T is denoted by $\mathcal{R}_\alpha(T)$ and $\mathcal{P}_\alpha(T)$, respectively. See Fig. 5.2 for an example of maximal gapped repeats and palindromes.

5.1 Related Work and Our Contribution

Most research articles on maximal α-gapped repeats and maximal α-gapped palindromes tackle the question of how many maximal α-gapped repeats/palindromes a text can have, and how to find them all efficiently (see Figure 5.3). Restrictions with respect to the gap and the domain of α lead to different approaches. The oldest contribution to this topic we are aware of is by Gusfield [120], who presented an algorithm computing the set of all maximal n-gapped repeats $\mathcal{R}_n(T)$ within $\mathcal{O}(n\sigma + |\mathcal{R}_n(T)|)$ time. He also presented an algorithm computing all gapped palindromes (i_L, i_R, u) with a fixed gap $i_R - i_L - u$, running in time linear in the length of the input times the alphabet size, plus the number of occurrences. Subsequently, Brodal et al. [37] presented an algorithm computing all α-gapped repeats in $\mathcal{O}(n \lg n)$ time under the restriction

All Maximal α-Gapped Repeats		
Time	Restriction	Ref.
$\mathcal{O}(n\sigma + \mathrm{occ})$	$\alpha = n$	[120, Sect. 7.12.3]
$\mathcal{O}(n\sigma + \mathrm{occ})$	$\alpha = n$	[2, Sect. 5.1]
$\mathcal{O}(n \lg n)$	$\alpha = \mathcal{O}(1)$	[37]
$\mathcal{O}(n \lg \alpha + \mathrm{occ})$	$g = \alpha$	[161]
$\mathcal{O}(\alpha^2 n)$	$\lvert\Sigma\rvert = \mathcal{O}(1)$	[166]
$\mathcal{O}(n \lg n + \alpha^2 n)$	$\lvert\Sigma\rvert = n^{\mathcal{O}(1)}$	[166]
$\mathcal{O}(\alpha n)$	$\lvert\Sigma\rvert = \mathcal{O}(1)$	[229]
$\mathcal{O}(\alpha n)$	$\lvert\Sigma\rvert = \mathcal{O}(1)$	[66]
$\mathcal{O}(\alpha n)$	$\lvert\Sigma\rvert = n^{\mathcal{O}(1)}$	Thm. 5.33
All Maximal α-Gapped Palindromes		
Time	Restriction	Ref.
$\mathcal{O}(n\sigma + \mathrm{occ})$	$g = \alpha$	[120, Sect. 9.2.2]
$\mathcal{O}(\alpha^2 n)$	$\lvert\Sigma\rvert = \mathcal{O}(1)$	[163]
$\mathcal{O}(\alpha n)$	$\lvert\Sigma\rvert = n^{\mathcal{O}(1)}$	Cor. 5.34

Fig. 5.3: Running time of algorithms computing all maximal α-gapped repeats (*top*) and palindromes (*bottom*) of the form $(i_\mathrm{L}, i_\mathrm{R}, u)$ with the gap $g := i_\mathrm{R} - i_\mathrm{L} - u$. The number of maximal α-gapped repeats (resp. palindromes) is denoted by $\mathrm{occ} = \mathcal{O}(\alpha n)$.

that $\alpha = \mathcal{O}(1)$. The approaches of Gusfield [120] and Brodal et al. [37] have the use of the suffix tree for the search in common. Without the suffix tree, Abouelhoda et al. [2] showed that $\mathcal{R}_n(T)$ can be computed with the suffix array and supporting data structures within the same time and space bounds as Gusfield [120]. Next, Kolpakov and Kucherov [163] introduced 2-gapped palindromes, and showed how to compute the set $\mathcal{P}_2(T)$ of all maximal 2-gapped palindromes in $\mathcal{O}(n + |\mathcal{P}_2(T)|)$ time for an input string T of length n over a constant alphabet. They left open the question of how large $|\mathcal{P}_2(T)|$ can actually be. In a follow-up study [166], the notion of α-gapped repeats for an arbitrary $\alpha \geq 1$ was coined. There, Kolpakov et al. [166] showed that the set $\mathcal{R}_\alpha(T)$ of all maximal α-gapped repeats can be computed in either (a) $\mathcal{O}(\alpha^2 n)$ time for constant alphabets or (b) $\mathcal{O}(\alpha^2 n + |\mathcal{R}_\alpha(T)|)$ time for integer alphabets. They further proved that $|\mathcal{R}_\alpha(T)| = \mathcal{O}(\alpha^2 n)$ for a text T of length n, and that the number of all maximal substrings with exponents in $(1 + \delta, 2]$ for a real number δ with $\delta \in (0, 1]$, so-called *δ-subrepetitions*, is bounded by the number of all maximal $1/\delta$-gapped repeats. In their preprint [165, Sect. 5], they posed two open problems concerning their computed bounds:

- developing a more efficient algorithm and
- closing the gap between the upper bound $\mathcal{O}(\alpha^2 n)$ and the lower bound $\Omega(\alpha n)$ (cf. Example 5.3) for the maximum number of all maximal α-gapped repeats.

These questions laid the foundation of the following line of research: The first

question was answered by Tanimura et al. [229] and Crochemore et al. [66] independently: each group of authors presented an $\mathcal{O}(\alpha n + |\mathcal{R}_\alpha(T)|)$-time algorithm computing all maximal α-gapped repeats in a string whose characters are drawn from a constant alphabet. The second question was answered to be order of αn [66], and subsequently to be at most $18\alpha n$ and $28\alpha n + 7n$ for all maximal α-gapped repeats and all maximal α-gapped palindromes, respectively [108]. Following this line of achievements, we give further improvements to those answers:

- The number of all maximal α-gapped repeats $|\mathcal{R}_\alpha(T)|$ in a text of length n is at most $3(\pi^2/6 + 5/2)\alpha n$ (Thm. 5.16).
- The number of all maximal α-gapped palindromes $|\mathcal{P}_\alpha(T)|$ in a text of length n is at most $7(\pi^2/6 + 1/2)\alpha n - 3n - 1$ (Thm. 5.22).
- We can compute $\mathcal{R}_\alpha(T)$ in $\mathcal{O}(\alpha n)$ time for integer alphabets (Thm. 5.33).
- The algorithm of Thm. 5.33 can be adapted to find $\mathcal{P}_\alpha(T)$ in $\mathcal{O}(\alpha n)$ time (Cor. 5.34).
- All results are valid when supporting and prohibiting overlaps of the arms.

While our combinatoric results build on the achievements of the point analysis described in [166, Lemma 6], our algorithmic ideas are derived from Dumitran et al. [70], who presented an algorithm computing the longest α-gapped repeat [70, Thm. 7] and the longest α-gapped palindrome [70, Thm. 8] in $\mathcal{O}(\alpha n)$ time.

The main difference to the former results is that we support overlaps (previous results assumed that $i_\mathrm{L} + u \leq i_\mathrm{R}$). This is a generalization because our proofs work for both supporting and prohibiting overlaps. It makes the maximality property more natural, since a left/right extension of a gapped repeat (resp. an inward extension of a gapped palindrome) is always a gapped repeat (resp. gapped palindrome).

Example 5.1. The first two characters of $T = \mathtt{aaa}$ form a gapped repeat $(1, 2, 1)$. The right extension $(1, 2, 2)$ of both arms is a gapped repeat only if overlaps are supported. Similarly, $(1, 3, 1)$ is a gapped palindrome, but the inward extension $(1, 2, 2)$ is a gapped palindrome only if overlaps are supported.

Including overlapping arms was already considered by Crochemore et al. [66], who split the combinatorial analysis up into maximal gapped repeats without overlaps (the gap $g := i_\mathrm{R} - i_\mathrm{L} - u$ is non-negative) and maximal gapped repeats with overlaps (g is negative). For the latter ones, they showed that the number of all gapped repeats with overlaps is less than n [66, Sect. 4]. This gives rise to the following lemma:

Lemma 5.2 ([66, Conclusions])**.** The number of all maximal 1-gapped repeats is less than $|T|$.

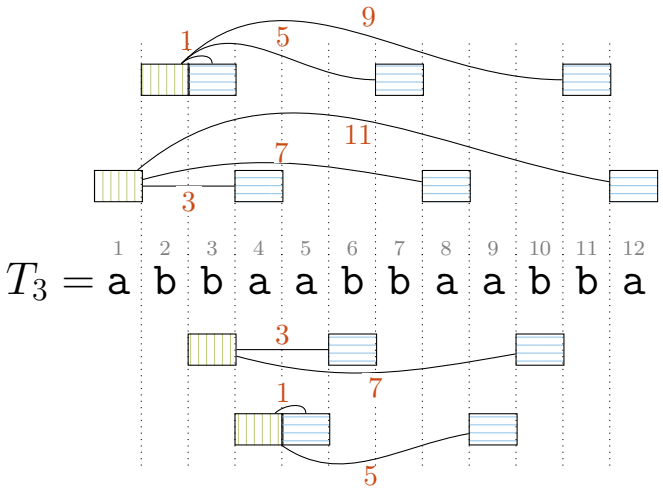


Fig. 5.4: Illustration of Example 5.3 with $k = 3$ depicting all maximal gapped repeats whose left arms (*vertically hatched rectangles* ▯) have length one and start within the first four positions. A left arm is connected with edges to all possible right arms (*horizontally hatched blocks* ▭). Each edge is labeled with the distance of the starting positions of the respective arms.

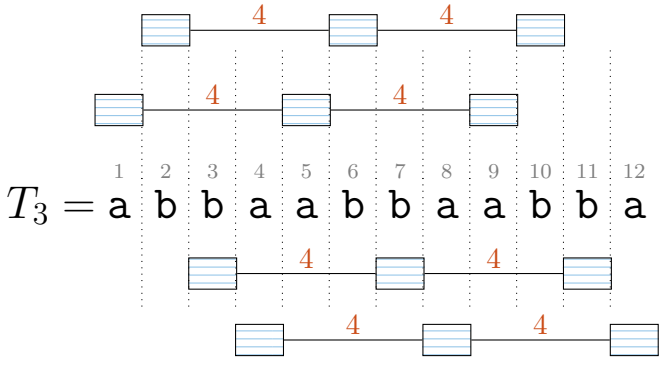


Fig. 5.5: Illustration of Example 5.3 with $k = 3$ depicting all maximal gapped palindromes with an arm length of one. Each pair of rectangles connected by a path P forms a maximal gapped palindrome, whose arm-period is the sum of the labels of the edges of P.

The following example shows that our obtained bounds on the number of all maximal α-gapped repeats and palindromes are asymptotically tight:

Example 5.3 ([66, directly after Thm. 2])**.** Given a real number α with $\alpha \geq 4$, the string $T_k \coloneqq (\mathtt{abba})^k$ with an integer $k \in \Omega(\alpha)$ contains $\Theta(\alpha k)$ maximal α-gapped repeats whose arms are of length one (see Fig. 5.4). Similarly, we find $\Theta(\alpha k)$ maximal α-gapped palindromes in T_k (see Fig. 5.5). We conclude that the number of maximal α-gapped repeats and the number of maximal α-gapped palindromes in the string T_k is $\Omega(\alpha k)$.

In the light of Example 5.3, we cannot hope for algorithms finding all α-gapped repeats or palindromes faster in the worst case. We conclude that the worst case running times of our algorithms are tight.

5.2 Preliminaries

When writing $T[b\,..\,e]$, we can mean two different things: the expression can denote both a substring and the occurrence of this substring starting at position b (and ending at position e) in T. The second entity is called the segment[1] $T[b\,..\,e]$. A *segment* $T[b\,..\,e]$ of a string T is the occurrence of a substring S equal to $T[b\,..\,e]$ in T; we say that S *occurs* at position b in T. While a substring is identified only by a sequence of characters, a segment is also identified by its position in the string. Consequently, segments are always unique, while a string may contain multiple occurrences of the same substring. We use the same notation to address substrings and segments of a string. For two segments S and $\overline{S}$ of a string T, we write $S \equiv \overline{S}$ if they start at the same position in T and have the same length. We write $S = \overline{S}$ if the substrings identifying these segments are the same (hence $S \equiv \overline{S} \Rightarrow S = \overline{S}$). We implicitly use segments both like substrings of T and as intervals contained in $[1\,..\,|T|]$, e.g., we write $S \subseteq \overline{S}$ if two segments $S \coloneqq T[b\,..\,e]$, $\overline{S} \coloneqq T[\overline{b}\,..\,\overline{e}]$ of T satisfy $[b\,..\,e] \subseteq [\overline{b}\,..\,\overline{e}]$, i.e., $\mathsf{b}(\overline{S}) \leq \mathsf{b}(S) \leq \mathsf{e}(S) \leq \mathsf{e}(\overline{S})$.

Two segments S and $\overline{S}$ of the same text T are called *consecutive* if $\mathsf{e}(S) + 1 = \mathsf{b}(\overline{S})$. Two occurrences S and $\overline{S}$ with $\mathsf{b}(S) < \mathsf{b}(\overline{S})$ of the same substring S in the text T are called *subsequent* if there is no occurrence of S starting between $\mathsf{b}(S) + 1$ and $\mathsf{b}(\overline{S}) - 1$.

5.2.1 Periodicity

The time spent on counting and computing α-gapped repeats and palindromes heavily depends on the repetitiveness of the text, i.e., the occurrences of periodic substrings in the text. To deal with periodic substrings, we build on the following classic lemma of Fine and Wilf:

[1] This notion was coined by Crochemore et al. [66].

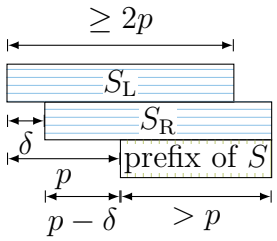


Fig. 5.6: Setting of the proof of Cor. 5.5 with $\delta < p$. There are two occurrences S_L and S_R of S with an overlap of $2p - \delta$ characters. Both occurrences induce a run with a period δ. There are at least three occurrences of S's prefix of length $p + 1$ (starting at $\mathsf{b}(S_\mathrm{L}), \mathsf{b}(S_\mathrm{R})$, and $\mathsf{b}(S_\mathrm{L}) + p$).

Lemma 5.4 ([80])**.** Given a string T with two periods p and p' such that $p + p' \leq |T|$, the greatest common divisor $\gcd(p, p')$ of p and p' is also a period of T.

Corollary 5.5. A periodic substring S in a text T with the smallest period p cannot have two distinct occurrences S_L and S_R in T with $|\mathsf{b}(S_\mathrm{L}) - \mathsf{b}(S_\mathrm{R})| < p$.

Proof. Since the smallest period of S is p, it holds that $|S| \geq 2p$. Assume for a contradiction that two distinct occurrences S_L and S_R of S exist in T with a distance of $\delta \coloneqq \mathsf{b}(S_\mathrm{R}) - \mathsf{b}(S_\mathrm{L})$ such that $0 < \delta < p$ (see also Fig. 5.6). Since $|S_\mathrm{L} \cap S_\mathrm{R}| \geq 2p - \delta \geq p$, the distance δ is a period of S. Additionally, since S has the smallest period p, there is another occurrence of a prefix of S starting at $\mathsf{b}(S_\mathrm{L}) + p - \delta$ with a length of at least $p + \delta > p$. Hence $p - \delta$ is also a period of S. Since the sum of both periods δ and $p - \delta$ is less than $|S|$, Lemma 5.4 states that $\gcd(\delta, p - \delta) < p$ is a period of S. This contradicts the fact that p is the smallest period of S. □

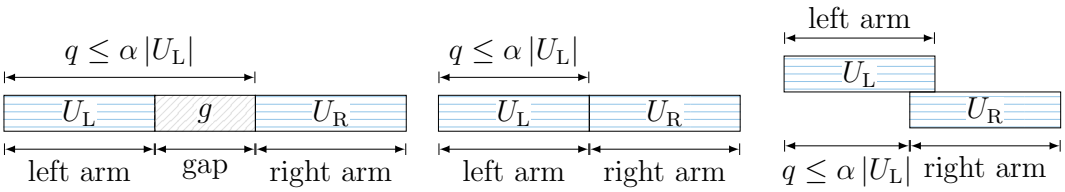
Corollary 5.6. The length of the overlap between two subsequent occurrences of an aperiodic substring S in a word T is upper bounded by $\lfloor |S| / 2 \rfloor$.

5.2.2 Gapped Repeats and Palindromes

Instead of working with triples of integers $(i_\mathrm{L}, i_\mathrm{R}, u)$ as at the beginning of this chapter (when introducing gapped repeats and palindromes), we switch notation in favor of the introduced segments to ease the analysis that follows. From now on we stick to pairs of segments $(T[i_\mathrm{L} \mathinner{..} i_\mathrm{L} + u - 1], T[i_\mathrm{R} \mathinner{..} i_\mathrm{R} + u - 1])$ (cf. Fig. 5.7): Given a text T, we call a pair of segments $(U_\mathrm{L}, U_\mathrm{R})$ a *gapped repeat* (resp. *gapped palindrome*) of T if

- $\mathsf{b}(U_\mathrm{L}) + 1 \leq \mathsf{b}(U_\mathrm{R})$ and $U_\mathrm{R} = U_\mathrm{L}$ in the case of a gapped repeat, or
- $\mathsf{b}(U_\mathrm{L}) \leq \mathsf{b}(U_\mathrm{R})$ and $U_\mathrm{R} = {U_\mathrm{L}}^\intercal$ in the case of a gapped palindrome (it is possible that $U_\mathrm{L} \equiv U_\mathrm{R}$).

The segments U_L and U_R are called left and right *arm*, respectively. The value $\mathsf{b}(U_\mathrm{R}) - \mathsf{e}(U_\mathrm{L}) - 1$ is called the *gap*. It is the distance between both arms in case that it is positive. The distance $q = \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$ is called the *arm-period* of



(a) $g > 0$ (b) $g = 0$ (c) $g < 0$

Fig. 5.7: Pairs of segments $(U_\mathrm{L}, U_\mathrm{R})$. A pair is called a gapped repeat if $0 < \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$ and $U_\mathrm{L} \equiv U_\mathrm{R}$. It is called a gapped palindrome if $0 \le \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$ and $U_\mathrm{R} \equiv U_\mathrm{L}^\intercal$. The segment $(U_\mathrm{L}, U_\mathrm{R})$ is called α-gapped if $\mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L}) \le \alpha\,|U_\mathrm{L}|$. The pair (a) has a positive gap $g := \mathsf{b}(U_\mathrm{R}) - \mathsf{e}(U_\mathrm{L}) - 1$, the pair (b) defines a square, and the segments U_L and U_R of pair (c) overlap.

$(U_\mathrm{L}, U_\mathrm{R})$, which must not be confused with the periods of periodic strings. Both terms have the following connection: The arm-period of a gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ with a non-positive gap is *a period* of the periodic substring $T[\mathsf{b}(U_\mathrm{L}) \mathinner{..} \mathsf{e}(U_\mathrm{R})]$. In particular, the arm-period of a gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ whose arms are consecutive (i.e., the gap is zero) is the *arm length* of the square $T[\mathsf{b}(U_\mathrm{L}) \mathinner{..} \mathsf{e}(U_\mathrm{R})]$.

For $\alpha \ge 1$, the gapped repeat or gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ is called *α-gapped* if its arm-period q is at most $\alpha\,|U_\mathrm{L}|$.

A gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ is called *maximal* if the characters to the immediate left and to the immediate right of its arms differ (as far as they exist), i.e.,

- $T[\mathsf{b}(U_\mathrm{L}) - 1] \ne T[\mathsf{b}(U_\mathrm{R}) - 1]$ (or $\mathsf{b}(U_\mathrm{L}) = 1$) and
- $T[\mathsf{e}(U_\mathrm{L}) + 1] \ne T[\mathsf{e}(U_\mathrm{R}) + 1]$ (or $\mathsf{e}(U_\mathrm{R}) = n$).

Similarly, a gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ is called *maximal* if it can be extended neither inwards nor outwards, i.e.,

- $T[\mathsf{b}(U_\mathrm{L}) - 1] \ne T[\mathsf{e}(U_\mathrm{R}) + 1]$ (or $\mathsf{b}(U_\mathrm{L}) = 1$ or $\mathsf{e}(U_\mathrm{R}) = n$) and
- $T[\mathsf{e}(U_\mathrm{L}) + 1] \ne T[\mathsf{b}(U_\mathrm{R}) - 1]$ (or $\mathsf{b}(U_\mathrm{R}) = 1$ or $\mathsf{e}(U_\mathrm{L}) = n$).

Let $\mathcal{R}_\alpha(T)$ (resp. $\mathcal{P}_\alpha(T)$) denote the set of all maximal α-gapped repeats (resp. palindromes) in T.

Gapped palindromes are a generalization of ordinary palindromes, since the left arm and the right arm of a gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ are ordinary palindromes if they coincide (i.e., $U_\mathrm{L} \equiv U_\mathrm{R}$); in such a case we say that the gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ is an *ordinary palindrome*. For a maximal gapped palindrome with a gap $\mathsf{b}(U_\mathrm{R}) - \mathsf{e}(U_\mathrm{L}) - 1 \le 1$ it follows that $U_\mathrm{L} \equiv U_\mathrm{R}$ (otherwise it could be extended inwards). Hence, all maximal gapped palindromes with a gap of at most one are maximal ordinary palindromes, and vice versa.

Figures 5.8 and 5.9 show examples for all α-gapped maximal repeats $\mathcal{R}_\alpha(T)$ and all α-gapped maximal palindromes $\mathcal{P}_\alpha(T)$, respectively. The representation

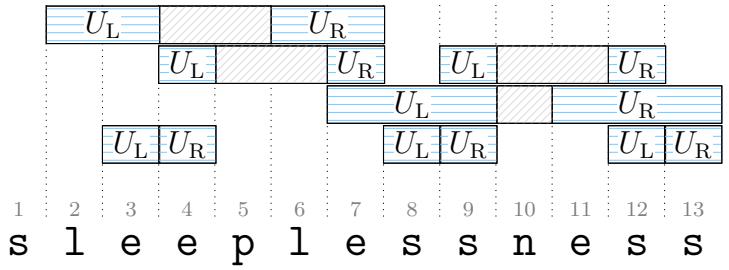


Fig. 5.8: All maximal α-gapped repeats of the string sleeplessness with $3 \leq \alpha < 5$.

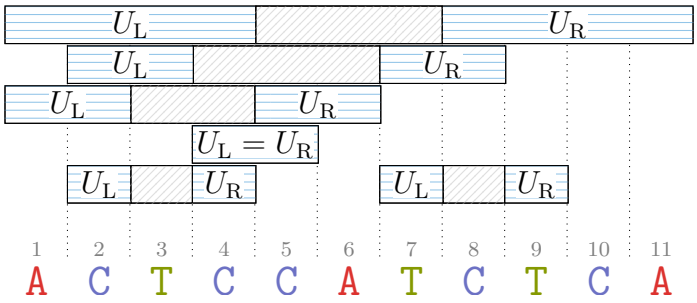


Fig. 5.9: All maximal α-gapped palindromes of the string ACTCCATCTCA with $\frac{5}{2} \leq \alpha < 3$.

of a maximal gapped repeat/palindrome by the segment $Z := T[\mathsf{b}(U_L) \,..\, \mathsf{e}(U_R)]$ is not unique — the same segment Z can be composed of gapped repeats/palindromes with different arm-periods. Instead, a maximal gapped repeat/palindrome is uniquely determined by its left arm U_L and its arm-period. For instance, the string $T =$ aabaa contains the maximal 4-gapped repeats $(T[1\,..\,2], T[4\,..\,5])$ and $(T[1], T[5])$, and both gapped repeats span over T.

Kolpakov et al. [166] split the set of all maximal α-gapped repeats into three subsets. They studied the maximal size of each subset:

- those whose arms are contained in one or two runs,
- those whose arms contain a periodic prefix or suffix larger than half of the size of the arms, and
- those belonging to neither of the two previous subsets.

They showed that the first two subsets contain at most $\mathcal{O}(\alpha n)$ elements combined. For the last subset, they applied the point analysis introduced in [158, Def. 17]. By mapping a gapped repeat to a point consisting of the end position of its left arm and its arm-period, they showed that the points created by two different maximal α-gapped repeats cannot $\frac{1}{4\alpha}$-cover the same point [166, Lemma 6]. With this property, they bounded the size of the last subset by $\mathcal{O}(\alpha^2 n)$. In Sect. 5.3.2, we present a refined version of this point analysis, with which we can show in Sect. 5.3.3 that the size of the last subset is $\mathcal{O}(\alpha n)$. As a consequence, the number of all maximal α-gapped repeats of a string of length n is $\mathcal{O}(\alpha n)$.

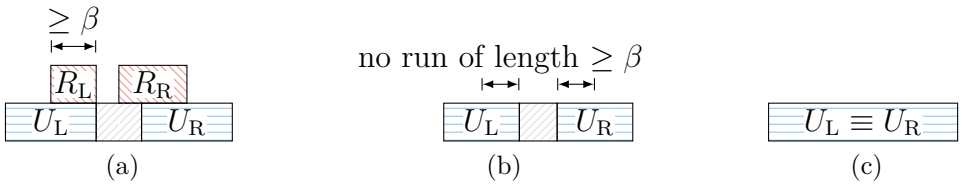


Fig. 5.10: Types of maximal α-gapped palindromes $(U_\mathrm{L}, U_\mathrm{R})$ under consideration. The segments R_L and R_R are runs. The gapped palindromes in Fig. (a) and Fig. (b) are *β-periodic* and *β-aperiodic*, respectively. The gapped palindrome in Fig. (c) is an ordinary palindrome.

5.3 Combinatoric Result

Unlike former approaches [66, 163, 166], we partition the set of all maximal α-gapped repeats (resp. palindromes) differently. We divide the set of all maximal α-gapped repeats $\mathcal{R}_\alpha(T)$ into two sets by the criterion whether their right arm contains a periodic prefix or not. The two subsets are analyzed differently: For the ones with a periodic prefix, we enumerate the runs covering this prefix. The other subset is analyzed with the results of Sect. 5.3.2.

We process the set of all maximal α-gapped palindromes $\mathcal{P}_\alpha(T)$ similarly, but additionally put all maximal ordinary palindromes in a separate, third subset[2], see also Fig. 5.10 for an overview. We begin with a formal definition of these subsets:

Given a positive number $\beta \in \mathbb{R}$, a maximal α-gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ (resp. a maximal α-gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ with $U_\mathrm{L} \not\equiv U_\mathrm{R}$) belongs to the set of all maximal α-gapped β-periodic repeats $\mathcal{R}^\beta_\alpha(T)$ (resp. palindromes $\mathcal{P}^\beta_\alpha(T)$) if U_R contains a periodic prefix of length at least $\beta\,|U_\mathrm{L}|$. We call the elements of $\mathcal{R}^\beta_\alpha(T)$ and $\mathcal{P}^\beta_\alpha(T)$ *β-periodic*. A maximal α-gapped repeat is called *β-aperiodic* if it is not β-periodic. A maximal α-gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ is called *β-aperiodic* if it is neither β-periodic nor a maximal ordinary palindrome. The set of all maximal α-gapped β-aperiodic repeats and the set of all maximal α-gapped β-aperiodic palindromes are denoted by $\overline{\mathcal{R}^\beta_\alpha}(T)$ and $\overline{\mathcal{P}^\beta_\alpha}(T)$, respectively. Figure 5.11 summarizes the partitioning of the sets $\mathcal{R}_\alpha(T)$ and $\mathcal{P}_\alpha(T)$.

5.3.1 β-Periodic Repeats and Palindromes

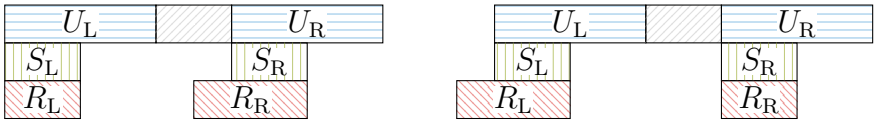
We start with an upper bound on the number of all maximal α-gapped β-periodic repeats $\mathcal{R}^\beta_\alpha(T)$ and palindromes $\mathcal{P}^\beta_\alpha(T)$ in the following lemma:

Lemma 5.7. Let T be a string. Further let α and β be two real numbers with $\alpha > 1$ and $0 < \beta < 1$. Then

[2] This subset is the set of all maximal α-gapped palindromes whose arms overlap.

Set	Properties	Bound	Attained in
$\mathcal{R}_\alpha^\beta(T)$	β-periodic repeats	$2\alpha\frac{\mathcal{E}(T)}{\beta}$	Lemma 5.7($\mathcal{R}$)
$\overline{\mathcal{R}_\alpha^\beta}(T)$	β-aperiodic repeats	$\left(\frac{\pi^2}{6}-\frac{1}{2}\right)\frac{\alpha n}{1-\beta}$	Cor. 5.15, $\frac{2}{3}\leq\beta<1$
$\mathcal{P}_\alpha^\beta(T)$	β-periodic pali.	$2(\alpha-1)\frac{\mathcal{E}(T)}{\beta}+2n$	Lemma 5.7($\mathcal{P}$),
$\overline{\mathcal{P}_\alpha^\beta}(T)$	β-aperiodic pali.	$\left(\frac{\pi^2}{6}-\frac{1}{2}\right)\frac{\alpha n}{1-\beta}$	Cor. 5.21, $\frac{7}{9}\leq\beta<1$
—	ordinary pali.	$2n-1$	Lemma 2.3

Fig. 5.11: Partitioning the set of all maximal α-gapped repeats $\mathcal{R}_\alpha(T)$ and the set of all maximal α-gapped palindromes $\mathcal{P}_\alpha(T)$. The column *Bound* lists the upper bound on the size of the respective set. The function $\mathcal{E}(T)$ denotes the sum of all exponents of all runs in T (see Chapter 2). Due to space restrictions *palindromes* is abbreviated to *pali.*.



(a) $\mathsf{b}(U_L)=\mathsf{b}(R_L)$ (b) $\mathsf{b}(U_R)=\mathsf{b}(R_R)$

Fig. 5.12: Setting of the proof of Lemma 5.7($\mathcal{R}$). Each figure shows a maximal α-gapped β-periodic repeat (U_L, U_R) and the periodic prefixes S_L and S_R of its respective arms U_L and U_R. The periodic prefixes are contained in the runs R_L and R_R, respectively. The equation (a) $\mathsf{b}(U_L)=\mathsf{b}(R_L)$ or (b) $\mathsf{b}(U_R)=\mathsf{b}(R_R)$ must hold. By the maximality property of runs, $\mathsf{e}(R_L)=\mathsf{e}(S_L)$ and $\mathsf{e}(R_R)=\mathsf{e}(S_R)$, i.e., $S_L\equiv R_L\cap U_L$ and $S_R\equiv R_R\cap U_R$.

($\mathcal{R}$) $\left|\mathcal{R}_\alpha^\beta(T)\right|$ is at most $2\alpha\,\mathcal{E}(T)/\beta$, and

($\mathcal{P}$) $\left|\mathcal{P}_\alpha^\beta(T)\right|$ is at most $2(\alpha-1)\,\mathcal{E}(T)/\beta+2n$.

Proof. Let $(U_L, U_R)\in\mathcal{R}_\alpha^\beta(T)$ (resp. $\in\mathcal{P}_\alpha^\beta(T)$) be a maximal α-gapped β-periodic repeat (resp. palindrome). By definition, the right arm U_R has a periodic prefix S_R of length at least $\beta\,|U_R|$. Let R_R denote the run that generates S_R, i.e., $S_R\subseteq R_R$. The two segments S_R and R_R have the smallest period p in common. By the definition of the gapped repeats (resp. palindromes), there is a prefix S_L of U_L (resp. suffix S_L of U_L) with $S_L=S_R$ (resp. $S_L=S_R{}^\intercal$). Let R_L be the run generating S_L. By definition, R_L has the same smallest period p as R_R.

($\mathcal{R}$) Gapped Repeats. Since (U_L, U_R) is maximal, $\mathsf{b}(U_L)=\mathsf{b}(R_L)$ or $\mathsf{b}(U_R)=\mathsf{b}(R_R)$ must hold (see Fig. 5.12); otherwise we could extend (U_L, U_R) to the left.

The periodic α-gapped repeat (U_L, U_R) is uniquely determined by its arm-period q and

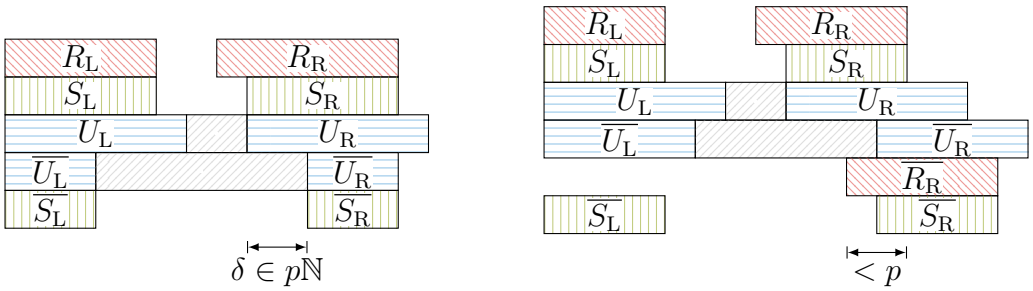


Fig. 5.13: Setting of the proof of Case (a) in Lemma 5.7($\mathcal{R}$) for two different maximal α-gapped β-periodic repeats (U_L, U_R) and $(\overline{U_L}, \overline{U_R})$ with $\mathsf{b}(U_L) = \mathsf{b}(\overline{U_L}) = \mathsf{b}(R_L)$. *Left:* The periodic prefixes S_R and $\overline{S_R}$ of the right arms of both gapped repeats are contained in a single run. The smallest period p of both runs R_L and R_R determines the possible starting positions of the right arms. *Right:* The periodic prefixes of the right arms of both gapped repeats are contained in different runs. Both runs cannot overlap more than $p-1$ positions due to Cor. 5.5.

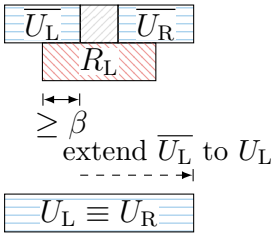


Fig. 5.14: A β-periodic gapped palindrome $(\overline{U_L}, \overline{U_R})$ with a run R_L covering the suffix (resp. prefix) of length β of the left arm $\overline{U_L}$ (resp. right arm $\overline{U_R}$). According to the proof of Lemma 5.7($\mathcal{P}$), we can make $(\overline{U_L}, \overline{U_R})$ maximal by extending it inwards to form an ordinary palindrome (U_L, U_R) with $U_L \equiv U_R$, which is of the same type as Fig. 5.10c.

(a) R_L in case $\mathsf{b}(U_L) = \mathsf{b}(R_L)$, or

(b) R_R in case $\mathsf{b}(U_R) = \mathsf{b}(R_R)$.

Since (U_L, U_R) is α-gapped, it holds that $q \leq \alpha u$ with $u := |U_L|$. We analyze Case (a), where $\mathsf{b}(U_L) = \mathsf{b}(S_L) = \mathsf{b}(R_L)$ holds. Case (b) is treated exactly in the same way by symmetry. The gapped repeat (U_L, U_R) is identified by its arm-period q and R_L. We fix R_L and pose the question of how many maximal periodic gapped repeats can be generated by R_L. We answer this question by counting the number of possible values for the arm-period q. Since the starting position $\mathsf{b}(S_R) = \mathsf{b}(U_R) = \mathsf{b}(U_L) + q = \mathsf{b}(R_L) + q$ of the periodic segment S_R is determined by q, two possible values of q must have a distance of at least p due to Cor. 5.5, see also Fig. 5.13.

It is left to find a lower and an upper bound for the value of q. For the lower bound, $q = \mathsf{b}(U_R) - \mathsf{b}(U_L) = \mathsf{b}(S_R) - \mathsf{b}(S_L)$ is at least $u \geq 2p$ in case that the arms U_L and U_R are not overlapping. Otherwise (in case the arms overlap), q is at least p because the starting positions of the periodic segments S_L and S_R have a distance of at least p (again due to Cor. 5.5). For the upper bound, with $u \leq |S_L| / \beta$ and $q \leq \alpha u$, we obtain $q \leq |S_L| \alpha/\beta \leq |R_L| \alpha/\beta$.

(a) $\mathsf{e}(U_L) = \mathsf{e}(R_L)$ (b) $\mathsf{b}(U_R) = \mathsf{b}(R_R)$

Fig. 5.15: Setting of the proof of Lemma 5.7($\mathcal{P}$). Each figure depicts a maximal α-gapped β-periodic palindrome (U_L, U_R) with the periodic suffix S_L. The periodic suffix $S_L \equiv R_L \cap U_L$ of U_L and the periodic prefix $S_R \equiv R_R \cap U_R$ of U_R are the intersections of the runs R_L and R_R with the respective arms. By the maximality property of runs, the equation (a) $\mathsf{e}(U_L) = \mathsf{e}(R_L)$ or (b) $\mathsf{b}(U_R) = \mathsf{b}(R_R)$ must hold.

Finally, we have that $p \le q \le |R_L|\,\alpha/\beta$. Then the number of possible arm-periods q is at most $|R_L|\,\alpha/(\beta p) = \exp(R_L)\alpha/\beta$. Overall, the number of all maximal α-gapped repeats is at most $\alpha\,\mathcal{E}(T)/\beta$ for the case $\mathsf{b}(U_L) = \mathsf{b}(R_L)$. Since Case (b) with $\mathsf{b}(U_R) = \mathsf{b}(R_R)$ is symmetric, we get the total upper bound $2\alpha\,\mathcal{E}(T)/\beta$.

($\mathcal{P}$) Gapped Palindromes. If $R_L \equiv R_R$ (see Fig. 5.14), then either $\mathsf{b}(U_R) - \mathsf{e}(U_L) \le 2$ (i.e., (U_L, U_R) is an ordinary palindrome), or (U_L, U_R) is not maximal. That is because of the following: Assume that R_L contains S_L and S_R. Then we have $T[\mathsf{e}(S_L) + 1] = T[\mathsf{e}(S_L) - p + 1] = T[\mathsf{b}(S_R) + p - 1] = T[\mathsf{b}(S_R) - 1]$, where the first and third equality follows from $|S_R| = |S_L| \ge 2p$, and the second equality follows from $S_R = S_L{}^{\mathsf{T}}$.

From now on, we assume that $R_L \not\equiv R_R$. Since (U_L, U_R) is maximal, $\mathsf{e}(U_L) = \mathsf{e}(R_L)$ or $\mathsf{b}(U_R) = \mathsf{b}(R_R)$ must hold; otherwise we could extend (U_L, U_R) inwards. This means that (U_L, U_R) is uniquely determined by the gap $g := \mathsf{b}(U_R) - \mathsf{e}(U_L) - 1$ and

(a) R_L in case $\mathsf{e}(U_L) = \mathsf{e}(R_L)$, or

(b) R_R in case $\mathsf{b}(U_R) = \mathsf{b}(R_R)$.

Since ordinary palindromes are excluded from the set of all maximal α-gapped β-periodic palindromes, the gap g is at least two. Cases (a) and (b) are depicted in Fig. 5.15.

We analyze Case (a) with $\mathsf{e}(S_L) = \mathsf{e}(R_L)$, Case (b) is treated exactly in the same way by symmetry. The gapped palindrome (U_L, U_R) is identified by its gap $g \ge 2$ and R_L. We fix R_L and count the number of possible values of g. Since the starting position $\mathsf{b}(S_R) = \mathsf{e}(R_L) + g + 1$ of the periodic segment S_R is determined by g, two possible values of g must have a distance of at least p due to Cor. 5.5, see also Fig. 5.16. Since $|U_L| \le |S_L|/\beta$ and (U_L, U_R) is α-gapped, $g \le (\alpha - 1)\,|U_L| \le (\alpha - 1)\,|S_L|/\beta$. Then the number of possible values for g is bounded by $1 + |S_L|\,(\alpha-1)/(\beta p) = 1 + |R_L|\,(\alpha-1)/(\beta p) = 1 + \exp(R_L)(\alpha-1)/\beta$.

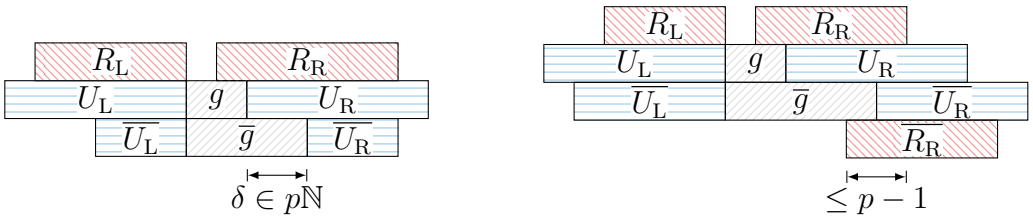


Fig. 5.16: Setting of Case (a) in the proof of Lemma 5.7($\mathcal{P}$) for two different maximal α-gapped β-periodic palindromes (U_L, U_R) and $(\overline{U_L}, \overline{U_R})$ with $\mathsf{e}(U_L) = \mathsf{e}(\overline{U_L}) = \mathsf{e}(R_L)$ and the respective gaps $g \coloneqq \mathsf{b}(U_R) - \mathsf{e}(U_L) - 1$ and $\overline{g} \coloneqq \mathsf{b}(\overline{U_R}) - \mathsf{e}(\overline{U_L}) - 1$. *Left:* The periodic prefixes of the right arms U_R and $\overline{U_R}$ of both gapped palindromes (U_L, U_R) and $(\overline{U_L}, \overline{U_R})$ are contained in a single run R_R. The smallest period p of R_R determines the possible starting positions of the right arms. *Right:* The periodic prefixes of the right arms of both gapped repeats are contained in different runs. Both runs cannot overlap more than $p-1$ positions due to Cor. 5.5.

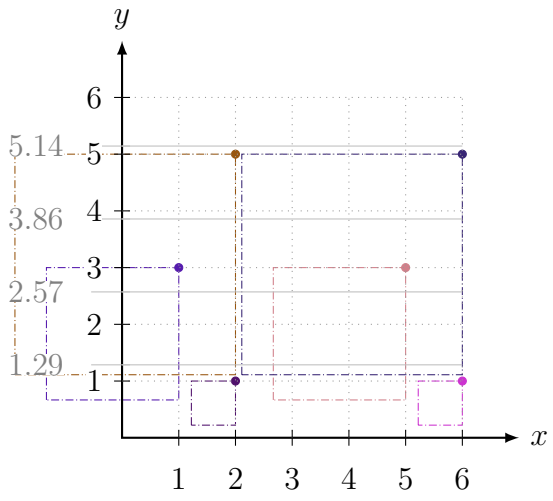


Fig. 5.17: 7/9-cover of the points $\{(4x - 2 - (y + 1 \mod 2), 2y - 1) \mid 1 \leq x, y \leq 3\} \subset \mathbb{N}^2$. The dash-dotted square of a point $\vec{p}$ comprises all points that are 7/9-covered by $\vec{p}$ (the square of $\vec{p}$ is the square that has $\vec{p}$ as its top right vertex). A point (x, y) with $y = 1$ only 7/9-covers itself. The light-gray dotted lines create the grid $\mathbb{N}^2$. For each integer i with $1 \leq i \leq 4$ the y-axis is label with the value i/γ with $\gamma \coloneqq 7/9$. Additionally, there is a gray horizontal line having the height i/γ.

In total, the number of maximal α-gapped palindromes in this case is bounded by $n + (\alpha - 1)\,\mathcal{E}(T)/\beta$ for the case $\mathsf{e}(U_L) = \mathsf{e}(R_L)$. Case (b) is symmetric, leading to the bound of $2(\alpha - 1)\,\mathcal{E}(T)/\beta + 2n$ in total. □

5.3.2 Improved Point Analysis

A pair of integers is called a *point*. We use points to upper bound the sizes of the sets $\overline{\mathcal{R}_\alpha^\beta}(T)$ and $\overline{\mathcal{P}_\alpha^\beta}(T)$ by mapping their elements to a set of points $C \subset \mathbb{Z}^2$. We estimate the size of C by the restriction that no two points in C γ-cover the same point in $\mathbb{Z}^2$. What γ-*cover* means is formally given in the following

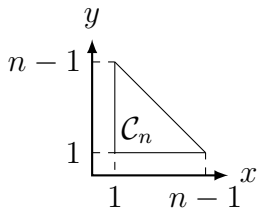
definition:

Definition 5.8. For a real number γ with $\gamma \in (0,1]$, we say that a point $(\hat{x},\hat{y}) \in \mathbb{Z}^2$ γ-*covers* a point $(x,y) \in \mathbb{Z}^2$ if $\hat{x} - \gamma\hat{y} \le x \le \hat{x}$ and $\hat{y}(1-\gamma) \le y \le \hat{y}$.

Note that the number of points that are γ-covered by a point $(\cdot, y)$ correlates with γ and the value y. Figure 5.17 gives an example for $\gamma := 7/9$.

For our purpose, it is sufficient to focus on the set

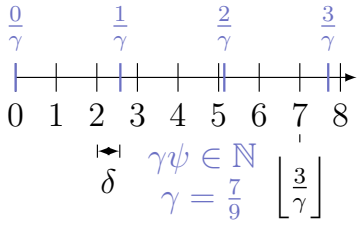
$$\mathcal{C}_n := \left\{(x,y) \in \mathbb{Z}^2 \mid 1 \le y \le n-1 \text{ and } 1 \le x \le n-y\right\},$$



since we will later show that we can map all maximal α-gapped repeats/palindromes to $\mathcal{C}_n$ injectively. Before that, we introduce two small helper lemmas that improve an inequality needed in Lemma 5.11:

Lemma 5.9. Given a real interval $\mathcal{I} := [\psi - 1/\gamma, \psi)$ with $\gamma, \psi \in \mathbb{R}$ and $\gamma \in (0,1]$,

$$|\mathcal{I} \cap \mathbb{Z}| = \begin{cases} \lfloor 1/\gamma \rfloor + 1 & \text{if } 0 < \psi - \lfloor \psi \rfloor \le \delta, \text{ or} \\ \lfloor 1/\gamma \rfloor & \text{otherwise,} \end{cases}$$



where $\delta := 1/\gamma - \lfloor 1/\gamma \rfloor$.

Proof. In the case that $\psi = \lfloor \psi \rfloor$ (i.e., $\psi \in \mathbb{Z}$), $\mathsf{b}(\mathcal{I}) = \psi - 1/\gamma \le \psi - \lfloor 1/\gamma \rfloor \in \mathcal{I} \cap \mathbb{Z}$. Hence $\{\psi - \lfloor 1/\gamma \rfloor, \ldots, \psi - 1\} = \mathcal{I} \cap \mathbb{Z}$, and $|\mathcal{I} \cap \mathbb{Z}| = \lfloor 1/\gamma \rfloor$.

In the case that $0 < \psi - \lfloor \psi \rfloor \le \delta$, we have $\psi - \delta \le \lfloor \psi \rfloor$, and therefore $\mathsf{b}(\mathcal{I}) = \psi - 1/\gamma = \psi - \lfloor 1/\gamma \rfloor - \delta \le \lfloor \psi \rfloor - \lfloor 1/\gamma \rfloor \in \mathcal{I} \cap \mathbb{Z}$. Hence $\{\lfloor \psi \rfloor - \lfloor 1/\gamma \rfloor, \ldots, \lfloor \psi \rfloor\} = \mathcal{I} \cap \mathbb{Z}$, and $|\mathcal{I} \cap \mathbb{Z}| = \lfloor 1/\gamma \rfloor + 1$ (because $\lfloor \psi \rfloor < \psi$).

The remaining case is that $\psi - \lfloor \psi \rfloor > \delta$. With $\lfloor \psi \rfloor < \psi - \delta = \psi - 1/\gamma + \lfloor 1/\gamma \rfloor$, we obtain that $\mathsf{b}(\mathcal{I}) = \psi - 1/\gamma > \lfloor \psi \rfloor - \lfloor 1/\gamma \rfloor \notin \mathcal{I} \cap \mathbb{Z}$. Hence $\{\lfloor \psi \rfloor - \lfloor 1/\gamma \rfloor + 1, \ldots, \lfloor \psi \rfloor\} = \mathcal{I} \cap \mathbb{Z}$, and $|\mathcal{I} \cap \mathbb{Z}| = \lfloor 1/\gamma \rfloor$. □

Lemma 5.10. Given the function $\nu_\gamma : \mathbb{N} \to \mathbb{N}$ with

$$\nu_\gamma(i) := |\{y \in \mathbb{N} \mid (i-1)/\gamma \le y < i/\gamma\}| \text{ for } i \in \mathbb{N},$$

and a non-increasing function $\mu : \mathbb{N} \to \mathbb{R}$, the inequality

$$\sum_{i=1}^{m} (\mu(i)\nu_\gamma(i)) \le \sum_{i=1}^{m} \mu(i)/\gamma \tag{5.1}$$

holds for every natural number m and every real number $\gamma \in (0,1]$.

Proof. We set $Y_i := \{y \in \mathbb{N} \mid (i-1)/\gamma \le y < i/\gamma\}$. Our task is to upper bound the size of Y_i since $\nu_\gamma(i) = |Y_i|$ for every $i \in \mathbb{N}$. It is clear that $|Y_i| \le \lfloor 1/\gamma \rfloor + 1$.

Since Y_1 cannot contain zero, it holds that $|Y_1| \leq \lfloor 1/\gamma \rfloor$ (if $1/\gamma \in \mathbb{N}$ then $|Y_1| = 1/\gamma - 1$, otherwise $|Y_1| = \lfloor 1/\gamma \rfloor$). For $i \geq 2$, Lemma 5.9 provides that

$$|Y_i| = \lfloor 1/\gamma \rfloor + 1 \text{ if and only if } 0 < i/\gamma - \lfloor i/\gamma \rfloor \leq \delta, \tag{5.2}$$

where $\delta \coloneqq 1/\gamma - \lfloor 1/\gamma \rfloor < 1$. Having Eq. (5.2), Eq. (5.1) is a conclusion of the following game estimating the cumulative sum of $\mu(i)/\gamma - \mu(i)\nu_\gamma(i)$: The game is divided in m rounds. In the i-th round ($1 \leq i \leq m$), we receive a credit of $(1/\gamma - \lfloor 1/\gamma \rfloor)\mu(i) = \delta\mu(i)$, but we additionally pay $\mu(i)$ from the credit when $\nu_\gamma(i) = \lfloor 1/\gamma \rfloor + 1$. If the credit does not become negative, it holds that $\sum_{i=1}^{m}(\mu(i)\nu_\gamma(i)) \leq \sum_{i=1}^{m} \mu(i)/\gamma$ (which is what we want to show in this proof).

Let $i_1, i_2, \ldots$ be the sequence of integers such that $\nu_\gamma(i_j) = \lfloor 1/\gamma \rfloor + 1$ for each j. After sorting this sequence ascendingly, it holds that $\delta i_j > j$ for every j. To see this, we write $i/\gamma - \lfloor i/\gamma \rfloor = i/\gamma - i\lfloor 1/\gamma \rfloor - \lfloor i/\gamma - i\lfloor 1/\gamma \rfloor \rfloor = \delta i - \lfloor \delta i \rfloor$, and apply Eq. (5.2): First, $\delta i_1 \geq 1$, since otherwise ($\delta i_1 < 1$) we obtain a contradiction to Eq. (5.2) with $\delta i_1 - \lfloor \delta i_1 \rfloor = \delta i_1 > 2\delta$ (remember that $i_1 \geq 2$ because $|Y_1| \leq \lfloor 1/\gamma \rfloor$). Next, assume that there exists a $j \geq 2$ such that $j \leq \delta i_j < \delta i_{j+1} < j + 1$. Then $\delta i_{j+1} - \lfloor \delta i_{j+1} \rfloor \geq \delta(i_j + 1) - \lfloor \delta i_j \rfloor > \delta$ (since $\delta i_j - \lfloor \delta i_j \rfloor > 0$), a contradiction that Eq. (5.2) holds for i_{j+1}. We conclude that $\delta i_j > j$ for every j.

Back to our game, we claim that there is at least $(\delta i_j - j)\mu(i_j)$ credit remaining after the i_j-th round. When reaching the i_1-th round, we have already gathered a credit of $\sum_{i=1}^{i_1} \delta\mu(i)$. Remember that we have to pay the amount $\mu(i_1)$. From our gathered credit we can pay $\mu(i_1)$ with $s \coloneqq \delta\mu(1) + \delta\mu(2) + \cdots + \delta\mu(i_1 - 1) + (1 - \delta(i_1 - 1))\mu(i_1)$: First, s is smaller than our gathered credit, since $\mu(i_1) < \delta i_1 \mu(i_1)$, and hence $(1 - \delta(i_1 - 1))\mu(i_1) < \delta\mu(i_1)$. Second, $s \geq \mu(i_1)$, because $\delta(i_1 - 1)\mu(i_1) \leq \sum_{i=1}^{i_1 - 1} \delta\mu(i)$ (remember that μ is non-increasing). By paying the amount s, a credit of at least $\mu(i_1)(\delta i_1 - 1)$ remains.

Under the assumption that our claim holds after the i_j-th round for an integer $j \in \mathbb{N}$, we show that the claim holds after the i_{j+1}-th round, too. According to our assumption, we have gathered a credit of at least $(\delta i_j - j)\mu(i_j) + \sum_{i=i_j+1}^{i_{j+1}} \delta\mu(i)$ at the beginning of the i_{j+1}-th round. We pay the amount $\mu(i_{j+1})$ with $s \coloneqq (\delta i_j - j)\mu(i_j) + \delta\mu(i_j + 1) + \cdots + \delta\mu(i_{j+1} - 1) + (j + 1 - \delta(i_{j+1} - 1))\mu(i_{j+1})$. First, s is smaller than our gathered credit, since $\delta i_{j+1} > j + 1$, and hence $(j + 1 - \delta(i_{j+1} - 1))\mu(i_{j+1}) < \delta\mu(i_{j+1})$. Second, $s \geq \mu(i_{j+1})$, because $\delta(i_{j+1} - 1)\mu(i_{j+1}) \leq (\delta i_j - j)\mu(i_j) + j\mu(i_{j+1}) + \sum_{i=i_j+1}^{i_{j+1}-1} \delta\mu(i)$. Similar to the i_1-th round, a credit of at least $(\delta i_{j+1} - j - 1)\mu(i_{j+1})$ remains. □

Lemma 5.11. Let γ be a real number with $\gamma \in (0, 1]$, and $C \subseteq \mathcal{C}_n$ be a set of points such that no two distinct points in C γ-cover the same point. Then $|C| < n\pi^2/(6\gamma)$. For $\gamma = 1$, we additionally obtain the bound $|C| \leq n(\pi^2/6 - 3/4)$.

Proof. Given that a point $\vec{p}$ in $\mathbb{Z}^2$ is γ-covered by a point $(\hat{x}, \hat{y})$ of C with $(i - 1)/\gamma \leq \hat{y} < i/\gamma$ for a positive integer i, we assign $\vec{p}$ the weight $1/i^2$. Otherwise ($\vec{p}$ is not γ-covered by any point of C), we assign $\vec{p}$ the weight zero.

Let us fix a point $(\hat{x}, \hat{y}) \in C$ with $(i-1)/\gamma \leq \hat{y} < i/\gamma$ for an integer i. We have $\hat{x} - i < \hat{x} - \gamma\hat{y} \leq \hat{x} - (i-1)$, and these inequalities also hold when substituting $\hat{x}$ with $\hat{y}$, i.e., $\hat{y} - i < \hat{y} - \gamma\hat{y} \leq \hat{y} - (i-1)$. There are exactly i^2 points $(x, y) \in \mathbb{Z}^2$ that are γ-covered by $(\hat{x}, \hat{y})$, since for each of them it holds that $\hat{x}-i < \hat{x}-\gamma\hat{y} \leq \hat{x}-(i-1) \leq x \leq \hat{x}$ and $\hat{y}-i < \hat{y}-\gamma\hat{y} \leq \hat{y}-(i-1) \leq y \leq \hat{y}$. Therefore, the sum of the weights of the points that are γ-covered by $(\hat{x}, \hat{y})$ is one. As a consequence, the size of C is equal to the sum of the weights of all points in $\mathbb{Z}^2$. In the following, let $\mathsf{weight}(\vec{p})$ denote the weight of a point $\vec{p}$. In what follows, we upper bound the sum of all weights.

First, we fix an integer y with $1 \leq y \leq n$, and show that the sum of the weights of all points $(\cdot, y)$ is at most n/i^2, where i is the integer with $(i-1)/\gamma \leq y < i/\gamma$. Given an integer $x \in \mathbb{Z}$, we conclude by the definition of $\mathcal{C}_n$ that

$$\mathsf{weight}(x, y) \begin{cases} \leq 1/i^2 & \text{for } 1 \leq x \leq n - y, \text{and} \\ = 0 & \text{for } x \geq n - y + 1. \end{cases}$$

The sum $\sum_{x=-\infty}^{1} \mathsf{weight}(x, y)$ is maximized to $1/i$ when each point in $E \coloneqq \{(x, y) \in \mathbb{Z}^2 \mid 2 - i \leq x \leq 1\}$ with $|E| = i$ has weight $1/i^2$, and the other points $\{(x, y) \in \mathbb{Z}^2 \mid x \leq 1 - i\}$ are not γ-covered. This can be seen by the following fact: A point (x, y) with $x \leq 1 - i$ can only be γ-covered by a point $(\hat{x}, \hat{y}) \in \mathcal{C}_n$ when $\hat{x} - \gamma\hat{y} \leq x \leq 1 - i$, or equivalently $i \leq \gamma\hat{y}$ (the smallest value for $\hat{x}$ is one). Assume that such a point $(\hat{x}, \hat{y})$ exists. Then there is an integer j with $i < j$ such that $\gamma\hat{y} < j$ and $(j-1)/\gamma \leq \hat{y} < j/\gamma$. Since $1 - j \leq \hat{x} - \gamma\hat{y} \leq x \leq 1$, there are at most $|\{(x, y) \mid 2 - j \leq x \leq 1\}| = j$ many different values for x. Furthermore, since $(\hat{x}, \hat{y}) \in \mathcal{C}_n$ γ-covers (x, y), it is not possible that another element of $\mathcal{C}_n$ γ-covers (x', y) with $x' < x$ (otherwise it would also cover (x, y)). In total, the sum under consideration $\sum_{x \leq 1} \mathsf{weight}(x, y)$ can be at most $1/j$, which is less than $1/i$. With $\sum_{x \leq 1} \mathsf{weight}(x, y) \leq 1/i$ we obtain $\sum_{x \in \mathbb{Z}} \mathsf{weight}(x, y) \leq (n - y - 1 + i)/i^2 \leq (n - y + \gamma y)/i^2 \leq n/i^2$.

Having computed $\sum_{x \in \mathbb{Z}} \mathsf{weight}(x, y)$ for a fixed y, we compute the sum over all y with $y \in \mathbb{Z}$. First, we deal with the special case that $\gamma = 1$. That is because it is the only case where $\mathsf{weight}(\cdot, 0)$ might not be zero (given $(\hat{x}, \hat{y}) \in \mathcal{C}_n$ and $\gamma < 1$, it holds that $\hat{y} \geq 1$ and therefore $0 < \hat{y} - \gamma\hat{y}$). A point (x, y) is 1-covered by $(\hat{x}, \hat{y}) \in \mathcal{C}_n$ if and only if $0 \leq y \leq \hat{y}$ and $\hat{x} - \hat{y} \leq x \leq \hat{y}$ hold. The weight of a point $(x, 0)$ with $0 \leq x \leq n - 1$ is maximized to $1/2^2$ if it is γ-covered by a point $(\hat{x}, \hat{y}) \in \mathcal{C}_n$ with the lowest possible value of $\hat{y}$, which is one. We conclude that $\sum_{x \in \mathbb{Z}} \mathsf{weight}(x, 0) \leq n/2^2$. With the same argument we conclude that $\sum_{x \in \mathbb{Z}} \mathsf{weight}(x, y) \leq n/(y+1)^2$ for every positive integer y. By summing everything up we obtain $\sum_{(x,y) \in \mathbb{Z}^2} \mathsf{weight}(x, y) \leq n/2^2 + n \sum_{y=1}^{n} (1/(y+1)^2) = n/4 + n \sum_{i=2}^{\infty} (1/i^2) = n/4 + n\pi^2/6 - n = n\pi^2/6 - 3n/4$ due to the Basel problem.

Finally, we consider the case that $\gamma < 1$. The idea is to cover the interval $[1 \,..\, n-1]$ with the sets $Y_i \coloneqq \{y \in \mathbb{N} \mid (i-1)/\gamma \leq y < i/\gamma\}$ for $1 \leq i \leq \lceil n\gamma \rceil$. Since a point (x, y_i) with $y_i \in Y_i$ has a weight of at most $1/i^2$, summing up all weights gives $\sum_{(x,y) \in \mathbb{Z}^2} \mathsf{weight}(x, y) \leq \sum_{i=1}^{\lceil n\gamma \rceil} n \left|Y_i\right| / i^2$. To compute $|Y_i|$, we use

the function $\nu_\gamma(i) \coloneqq |Y_i|$ as defined in Lemma 5.10. With ν_γ the upper bound of $\sum_{(x,y)\in\mathbb{Z}^2} \mathbf{weight}(x,y)$ can be stated as $\sum_{i=1}^{\lceil n\gamma\rceil}(\nu_\gamma(i)n/i^2)$. Since $\nu_\gamma(i) \leq \lfloor 1/\gamma\rfloor + 1$, it is easy to see that $\sum_{i=1}^{\lceil n\gamma\rceil}(\nu_\gamma(i)n/i^2) < n(\lfloor 1/\gamma\rfloor + 1)\sum_{i=1}^{\lceil n\gamma\rceil}(1/i^2) < n(\lfloor 1/\gamma\rfloor + 1)\pi^2/6$. By defining the non-increasing function μ with $\mu(i) \coloneqq n/i^2$, Lemma 5.10 yields (set $m = \lceil n\gamma\rceil$) $\sum_{i=1}^{\lceil n\gamma\rceil}(\nu_\gamma(i)n/i^2) = \sum_{i=1}^{\lceil n\gamma\rceil}\nu_\gamma(i)\mu(i) \leq (n/\gamma)\sum_{i=1}^{\lceil n\gamma\rceil}(1/i^2) < \sum_{i=1}^{\infty} n/(\gamma i^2) = n\pi^2/(6\gamma)$, which is also an upper bound of $|C|$. □

By restricting the subset $C \subseteq \mathcal{C}_n$ in Lemma 5.11 to be additionally bijective to the set of all maximal α-gapped repeats or palindromes, we can refine the upper bound attained in Lemma 5.11. How we do this is shown in Sects. 5.3.3 and 5.3.4:

5.3.3 β-Aperiodic Repeats

For the maximal α-gapped repeats, we follow Kolpakov et al. [166] who map a maximal α-gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ with arm-period $q \coloneqq \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$ to $(\mathsf{e}(U_\mathrm{L}), q)$. It holds that $(\mathsf{e}(U_\mathrm{L}), q) \in \mathcal{C}_n$, because $\mathsf{e}(U_\mathrm{R})$ and q are positive, and $\mathsf{e}(U_\mathrm{L}) + q = \mathsf{e}(U_\mathrm{R}) \leq n$. In particular we have $\mathsf{e}(U_\mathrm{L}) \leq n-1$, since otherwise $(\mathsf{e}(U_\mathrm{L}) = n)$ both endings $\mathsf{e}(U_\mathrm{R})$ and $\mathsf{e}(U_\mathrm{L})$ would be equal, and therefore $U_\mathrm{L} \equiv U_\mathrm{R}$ (a contradiction to the definition of gapped repeats). Let $\xi_\mathcal{R}$ denote the mapping from $(U_\mathrm{L}, U_\mathrm{R})$ to $(\mathsf{e}(U_\mathrm{L}), q)$, and let

$$\xi_\mathcal{R}(\mathcal{R}_\alpha(T)) \coloneqq \{\xi_\mathcal{R}(U_\mathrm{L}, U_\mathrm{R}) \mid (U_\mathrm{L}, U_\mathrm{R}) \text{ is a maximal } \alpha\text{-gapped repeat}\} \subset \mathcal{C}_n$$

denote the image of $\xi_\mathcal{R}$. The following lemma bounds the size of $\xi_\mathcal{R}(\mathcal{R}_\alpha(T))$ to be roughly at half of the size of $\mathcal{C}_n$, a fact that is used in Lemma 5.13.

Lemma 5.12. If $(x,y) \in \xi_\mathcal{R}(\mathcal{R}_\alpha(T))$, then $(x+1, y) \notin \xi_\mathcal{R}(\mathcal{R}_\alpha(T))$.

Proof. Let $(U_\mathrm{L}, U_\mathrm{R})$ be a maximal α-gapped β-aperiodic repeat with arm-period $q = \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$, and $(x,y) \coloneqq \xi_\mathcal{R}(U_\mathrm{L}, U_\mathrm{R}) = (\mathsf{e}(U_\mathrm{L}), q)$. If $(x+1, y) \in \xi_\mathcal{R}(\mathcal{R}_\alpha(T))$, then $T[x+1] = T[\mathsf{e}(U_\mathrm{L}) + 1] = T[x+y+1] = T[\mathsf{e}(U_\mathrm{R}) + 1]$, which contradicts the maximality of $(U_\mathrm{L}, U_\mathrm{R})$. □

With Lemma 5.12 we attain a version of Lemma 5.11 tailored to subsets of $\xi_\mathcal{R}(\mathcal{R}_\alpha(T))$:

Lemma 5.13. Let γ be a real number with $\gamma \in (0,1]$. A set of points $C \subseteq \xi_\mathcal{R}(\mathcal{R}_\alpha(T))$ such that no two distinct points in C γ-cover the same point obeys the inequality $|C| < n(\pi^2/6 - 1/2)/\gamma$.

Proof. If $\gamma = 1$, Lemma 5.11 already gives $|C| < n\pi^2/6 - 3n/4 < n\pi^2/6 - n/2$. For the case $\gamma < 1$, we focus on the points

$$E \coloneqq \{(x,y) \mid 1 \leq x \leq n \text{ and } y < 1/\gamma\}\,.$$

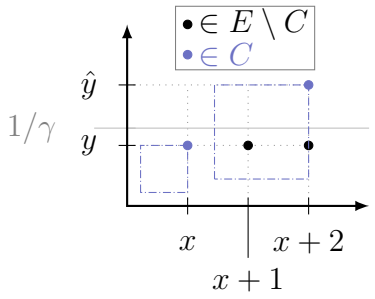


Fig. 5.18: Setting of the proof of Lemma 5.13, where the point $(x, y) \in C$, but $(x+1, y) \notin C$ with $\mathbf{weight}(x+1, y) > 0$. Thus $(x+1, y)$ is γ-covered by a point $(\hat{x}, \hat{y}) \in C$ ($\hat{x} = x+2$ in this figure). Like in Fig. 5.17, the dash-dotted rectangle of a point $\vec{p} \in C$ comprises all points that are γ-covered by $\vec{p}$. The points that are γ-covered by $(\hat{x}, \hat{y})$ are contained in the top right dashed rectangle. It can be seen that $(x+2, y)$ is also γ-covered by $(\hat{x}, \hat{y})$, and therefore cannot be in C.

In the proof of Lemma 5.11, we used the weights $\mathbf{weight}(\vec{p})$ of all points $\vec{p} \in \mathbb{Z}^2$ as an upper of $|C|$. There, we bounded the sum of the weights of all points in E by n/γ (assign each point the weight 1). We can refine this upper bound by halving the weights of the points in E. We justify this with the following analysis.

First, each point $(n, y) \in E$ has weight zero, since there is no point $(\hat{x}, \hat{y}) \in C$ (and even in $\mathcal{C}_n$) with $n \leq \hat{x}$. Thus the sum of the weights of the points $(n-1, y)$ and (n, y) is at most one, for every y with $1 \leq y < 1/\gamma$.

Second, a point $(x, y) \in E \cap C$ can only cover itself, since $y < 1/\gamma$. Consequently, a point $(x, y) \in E \setminus C$ can have a weight of at most $1/2^2 = 1/4$, since all points $(\hat{x}, \hat{y}) \in C \setminus E$ have $\hat{y} \geq 1/\gamma$. Given that $E \cap C = \emptyset$, the total weight of all points in E is at most $(1/4)\,|E|$.

Finally, suppose there is a point $(x, y) \in E \cap C$. Then $\mathbf{weight}(x, y) = 1$. Given $x \leq n-2$, $(x+1, y) \in \mathcal{C}_n$, but $(x+1, y) \notin C$ according to Lemma 5.12. We consider two cases:

- $\mathbf{weight}(x+1, y) = 0$. Then both points (x, y) and $(x+1, y)$ together have a weight of one.
- $\mathbf{weight}(x+1, y) > 0$, see also Fig. 5.18. Since $(x+1, y) \notin C$, $\mathbf{weight}(x+1, y) \leq 1/4$, i.e., it is γ-covered by a point $(\hat{x}, \hat{y}) \in C \setminus E$. Since $\hat{y} \geq 1/\gamma$, the point $(\hat{x}, \hat{y})$ γ-covers at least four points (including itself). Since $\mathbf{weight}(x, y) = 1$, $(\hat{x}, \hat{y})$ cannot γ-cover (x, y). Instead, it γ-covers the point $(x+2, y)$. We conclude that $(x+2, y) \notin C$. All three points (x, y), $(x+1, y)$, and $(x+2, y)$ have a total weight of at most $1 + 1/4 + 1/4 = 3/2$.

In both cases, a node has the average weight of at most $1/2$. Summing up all average weights yields the total weight of all points in E, which is at most $(1/2)\,|E| = n/(2\gamma)$.

Following the proof of Lemma 5.11, our modification of the weights changes the non-increasing function μ, which is now defined by $\mu(1) \coloneqq n/2$ and $\mu(i) \coloneqq n/i^2$ for $i \geq 2$. Changing μ yields the upper bound $\sum_{i=1}^{\lceil n\gamma \rceil} (\mu(i)\nu_\gamma(i)) \leq n(1/2 + \sum_{i=2}^{\lceil n\gamma \rceil} (1/i^2))/\gamma < n(\pi^2/6 - 1/2)/\gamma$ on the size of C. □

Finally, we want to apply the result of Lemma 5.13 to the set $\overline{\mathcal{P}_\alpha^\beta}(T)$. By restricting the domain of β, we can show that no pair of points of this set can $(1-\beta)/\alpha$-cover the same point:

Lemma 5.14. Given a string T, and two real numbers α, β with $\alpha > 1$ and $2/3 \leq \beta < 1$, the points mapped by two different maximal β-aperiodic α-gapped repeats in $\overline{\mathcal{R}_\alpha^\beta}(T)$ cannot $\frac{1-\beta}{\alpha}$-cover the same point.

Proof. Let $(U_\mathrm{L}, U_\mathrm{R})$ and $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ be two different maximal α-gapped β-aperiodic repeats in $\overline{\mathcal{R}_\alpha^\beta}(T)$. Set $u \coloneqq |U_\mathrm{L}| = |U_\mathrm{R}|$, $\overline{u} \coloneqq \left|\overline{U_\mathrm{L}}\right| = \left|\overline{U_\mathrm{R}}\right|$, $q \coloneqq \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$ and $\overline{q} \coloneqq \mathsf{b}(\overline{U_\mathrm{R}}) - \mathsf{b}(\overline{U_\mathrm{L}})$. We map the maximal gapped repeats $(U_\mathrm{L}, U_\mathrm{R})$ and $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ to the points $\xi_\mathcal{R}(U_\mathrm{L}, U_\mathrm{R}) = (\mathsf{e}(U_\mathrm{L}), q)$ and $\xi_\mathcal{R}(U_\mathrm{L}, U_\mathrm{R}) = (\mathsf{e}(\overline{U_\mathrm{L}}), \overline{q})$, respectively. Assume, for the sake of contradiction, that both points $\frac{1-\beta}{\alpha}$-cover the same point (x, y).

Let $\zeta \coloneqq \left|\mathsf{e}(U_\mathrm{L}) - \mathsf{e}(\overline{U_\mathrm{L}})\right|$ be the difference of the endings of both left arms, and $S_\mathrm{L} \coloneqq T[[\mathsf{b}(U_\mathrm{L}) \mathinner{..} \mathsf{e}(U_\mathrm{L})] \cap [\mathsf{b}(\overline{U_\mathrm{L}}) \mathinner{..} \mathsf{e}(\overline{U_\mathrm{L}})]]$ be the overlap of U_L and $\overline{U_\mathrm{L}}$. Let $s \coloneqq |S_\mathrm{L}|$, and let S_R (resp. $\overline{S_\mathrm{R}}$) be the right copy of S_L based on $(U_\mathrm{L}, U_\mathrm{R})$ (resp. $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$).

Sub-Claim. The overlap S_L is not empty, and $\mathsf{b}(S_\mathrm{R}) \neq \mathsf{b}(\overline{S_\mathrm{R}})$ (i.e., $S_\mathrm{R} \not\equiv \overline{S_\mathrm{R}}$).

Sub-Proof. Assume for this sub-proof that $\mathsf{e}(U_\mathrm{L}) < \mathsf{e}(\overline{U_\mathrm{L}})$ (otherwise exchange $(U_\mathrm{L}, U_\mathrm{R})$ with $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$, or yield the contradiction that $U_\mathrm{L} \equiv \overline{U_\mathrm{L}}$ and $U_\mathrm{R} \equiv \overline{U_\mathrm{R}}$). The latter contradiction ($U_\mathrm{L} \equiv \overline{U_\mathrm{L}}$ and $U_\mathrm{R} \equiv \overline{U_\mathrm{R}}$) is due to the following consideration: Since $\mathsf{e}(U_\mathrm{L}) = \mathsf{e}(\overline{U_\mathrm{L}})$, S_L cannot be empty (it is the intersection of both left arms). Further, both right copies are defined as the right translation of S_L by q and $\overline{q}$, respectively. If both right copies are identical, then $q = \overline{q}$, which contradicts the fact that $\xi_\mathcal{R}$ is injective.

Having $\mathsf{e}(U_\mathrm{L}) < \mathsf{e}(\overline{U_\mathrm{L}})$, we can combine (a) the $(1-\beta)/\alpha$-cover property with (b) the inequality $\beta < 1$ and (c) the fact that $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ is α-gapped, and yield $\mathsf{e}(\overline{U_\mathrm{L}}) - \overline{u} \leq_{(\mathrm{a})} \mathsf{e}(\overline{U_\mathrm{L}}) - \overline{q}/\alpha <_{(\mathrm{b})} \mathsf{e}(\overline{U_\mathrm{L}}) - \overline{q}(1-\beta)/\alpha \leq_{(\mathrm{c})} x \leq_{(\mathrm{c})} \mathsf{e}(U_\mathrm{L}) < \mathsf{e}(\overline{U_\mathrm{L}})$. Hence, the segment $T[\mathsf{e}(U_\mathrm{L})]$ is contained in $\overline{U_\mathrm{L}}$. If $S_\mathrm{R} \equiv \overline{S_\mathrm{R}}$, then we get a contradiction to the maximality of $(U_\mathrm{L}, U_\mathrm{R})$: By the above inequality, $T[\mathsf{e}(U_\mathrm{L}) + 1]$ is contained in $\overline{U_\mathrm{L}}$, too. Since $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ is a gapped repeat, the character $T[\mathsf{e}(U_\mathrm{L}) + 1]$ occurs in $\overline{U_\mathrm{R}}$, exactly at $T[\mathsf{e}(U_\mathrm{R}) + 1]$. ■

This sub-claim shows that $q \neq \overline{q}$. Without loss of generality let $q < \overline{q}$. Due to the $(1-\beta)/\alpha$-cover property it holds that

$$\overline{q} - \frac{\overline{q}(1-\beta)}{\alpha} \leq y \leq q \leq \overline{q}. \tag{5.3}$$

The difference of both arm-periods $\delta \coloneqq \overline{q} - q$ is

$$0 < \delta \leq \overline{q}(1-\beta)/\alpha \leq \overline{u}(1-\beta). \tag{5.4}$$

Equation (5.3) also yields that

$$u \geq q/\alpha \geq \frac{\overline{q}}{\alpha}\left(1 - \frac{1-\beta}{\alpha}\right) \geq \overline{q}\beta/\alpha. \tag{5.5}$$

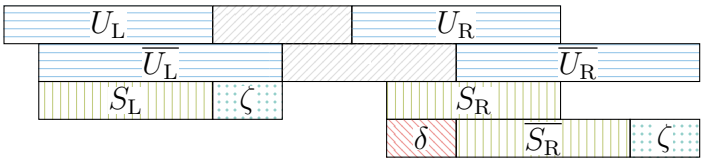
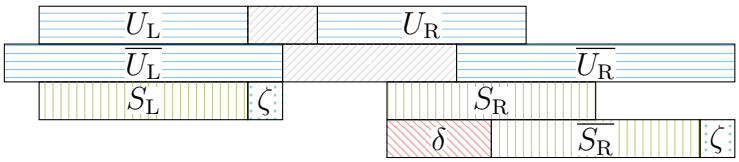


Fig. 5.19: Sub-Case 1a in the proof of Lemma 5.14 with $\mathsf{b}(U_L) \leq \mathsf{b}(\overline{U_L}) \leq \mathsf{e}(U_L) \leq \mathsf{e}(\overline{U_L})$.

Fig. 5.20: Sub-Case 1b in the proof of Lemma 5.14 with $\mathsf{b}(\overline{U_L}) < \mathsf{b}(U_L) \leq \mathsf{e}(U_L) \leq \mathsf{e}(\overline{U_L})$.



Since $S_R = [\mathsf{b}(S_L) + q \,..\, \mathsf{e}(S_L) + q]$ and $\overline{S_R} = [\mathsf{b}(S_L) + \overline{q} \,..\, \mathsf{e}(S_L) + \overline{q}]$, we have $\mathsf{b}(\overline{S_R}) - \mathsf{b}(S_R) = \delta$.

In the following we conduct a case analysis on the relationship between the starting and ending positions of the left arms of both gapped repeats. In each case, we show that the length s of the overlap of both arms S_L is at least 2δ, which means that the right copies S_R and $\overline{S_R}$ overlap with at least half of their common length. The overlap causes S_L to be periodic, which is a prefix of U_L or $\overline{U_L}$. If s is sufficiently large ($s \geq \beta u$ or $s \geq \beta \overline{u}$), the analysis leads to the contradiction that (U_L, U_R) or $(\overline{U_L}, \overline{U_R})$ are in $\mathcal{R}_\alpha^\beta(T)$. To show that s is sufficiently large, we use the fact that either (a) one left arm is contained in the other (and hence $S_L \equiv U_L$ or $S_L \equiv \overline{U_L}$) or that (b) the difference of the ends of both arms is upper bounded (because $u = s + \zeta$ or $\overline{u} = s + \zeta$).

Case 1: $\mathsf{e}(U_L) \leq \mathsf{e}(\overline{U_L})$. Since $\mathsf{e}(\overline{U_L}) - \overline{q}(1-\beta)/\alpha \leq x \leq \mathsf{e}(U_L) \leq \mathsf{e}(\overline{U_L})$,

$$\zeta = \mathsf{e}(\overline{U_L}) - \mathsf{e}(U_L) \leq \overline{q}(1-\beta)/\alpha \leq \overline{u}(1-\beta). \tag{5.6}$$

Sub-Case 1a: $\mathsf{b}(U_L) \leq \mathsf{b}(\overline{U_L})$, see Fig. 5.19. By Eq. (5.6), we get $s = \overline{u} - \zeta \geq \overline{u}\beta$. It follows from Eq. (5.4) and $\beta \geq 2/3$ that $s/\delta \geq \overline{u}\beta/\overline{u}(1-\beta) = \beta/(1-\beta) \geq 2$, which means that S_R and $\overline{S_R}$ overlap at least half of their common length. This overlap makes S_L periodic. Since S_L is a prefix of $\overline{U_L}$ of length $s \geq \overline{u}\beta$, $(\overline{U_L}, \overline{U_R})$ is in $\mathcal{R}_\alpha^\beta(T)$, a contradiction.

Sub-Case 1b: $\mathsf{b}(U_L) > \mathsf{b}(\overline{U_L})$, see Fig. 5.20. We conclude that $S_L = U_L$. It follows from Eqs. (5.4) and (5.5) and $2/3 \leq \beta < 1$ that $s/\delta \geq \overline{q}\alpha\beta/(\overline{q}\alpha(1-\beta)) = \beta/(1-\beta) \geq 2$, which means that $S_L = U_L$ is periodic. Hence (U_L, U_R) is in $\mathcal{R}_\alpha^\beta(T)$, a contradiction.

Case 2: $\mathsf{e}(U_L) > \mathsf{e}(\overline{U_L})$. Since $\mathsf{e}(U_L) - q(1-\beta)/\alpha \leq x \leq \mathsf{e}(\overline{U_L}) \leq \mathsf{e}(U_L)$,

$$\zeta = \mathsf{e}(U_L) - \mathsf{e}(\overline{U_L}) \leq q(1-\beta)/\alpha \leq \overline{q}(1-\beta)/\alpha \leq \overline{u}(1-\beta). \tag{5.7}$$

Sub-Case 2a: $\mathsf{b}(U_L) \leq \mathsf{b}(\overline{U_L})$, see Fig. 5.21. We conclude that $S_L = \overline{U_L}$. It follows from Eq. (5.4) and $\beta \geq 2/3$ that $s/\delta \geq \overline{u}/(\overline{u}(1-\beta)) = 1/(1-\beta) \geq 3 > 2$, which means that $S_L = \overline{U_L}$ is periodic. Hence $(\overline{U_L}, \overline{U_R})$ is in $\mathcal{R}_\alpha^\beta(T)$, a contradiction.

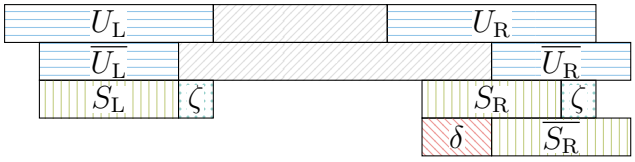
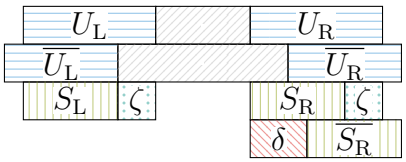


Fig. 5.21: Sub-Case 2a in the proof of Lemma 5.14 with $\mathsf{b}(U_L) \leq \mathsf{b}(\overline{U_L}) \leq \mathsf{e}(\overline{U_L}) < \mathsf{e}(U_L)$.

Fig. 5.22: Sub-Case 2b in the proof of Lemma 5.14 with $\mathsf{b}(\overline{U_L}) < \mathsf{b}(U_L) \leq \mathsf{e}(\overline{U_L}) < \mathsf{e}(U_L)$. The second inequality holds because the overlap S_L cannot be empty due to the sub-claim.



Sub-Case 2b: $\mathsf{b}(U_L) > \mathsf{b}(\overline{U_L})$, see Fig. 5.22. By Eq. (5.7) we get $\zeta \leq q(1-\beta)/\alpha \leq u(1-\beta)$ and hence $s = u - \zeta \geq u\beta$. By Eqs. (5.4) and (5.5) we get $u/\delta \geq \overline{q}\alpha\beta/(\overline{q}\alpha(1-\beta)) = \beta/(1-\beta) \geq 2$ with $\beta \geq 2/3$. Hence δ is upper bounded by $u/2$. This means that U_R has a periodic prefix of length at least 2δ (since $2\delta > s \geq u\beta$), a contradiction. □

The next corollary follows immediately from Lemmas 5.13 and 5.14.

Corollary 5.15. Given two real numbers α, β with $\alpha > 1$ and $2/3 \leq \beta < 1$, the number of all maximal α-gapped β-aperiodic repeats $\left|\overline{\mathcal{R}^{\beta}_{\alpha}(T)}\right|$ of a string T of length n is less than $(\pi^2/6 - 1/2)\alpha n/(1-\beta)$.

This result yields our final result on the maximum number of all maximal α-gapped repeats. The result is summarized in the following theorem:

Theorem 5.16. Given a real number α with $\alpha > 1$ and a text T of length n, the number of all α-gapped repeats $|\mathcal{R}_{\alpha}(T)|$ is less than $3(\pi^2/6 + 5/2)\alpha n$.

Proof. Combining the results of Lemma 5.7($\mathcal{R}$) and Cor. 5.15 yields that $|\mathcal{R}_{\alpha}(T)| = \left|\mathcal{R}^{\beta}_{\alpha}(T)\right| + \left|\overline{\mathcal{R}^{\beta}_{\alpha}(T)}\right| < 2\alpha\,\mathcal{E}(T)/\beta + (\pi^2/6 - 1/2)\alpha n/(1-\beta)$ for $2/3 \leq \beta < 1$. This number becomes minimal with $3\alpha\,\mathcal{E}(T) + 3(\pi^2/6 - 1/2)\alpha n$ when setting β to $2/3$. With Lemma 2.1 we obtain $|\mathcal{R}_{\alpha}(T)| < 9\alpha n + 3(\pi^2/6 - 1/2)\alpha n = 3(\pi^2/6 + 5/2)\alpha n$. □

With Lemma 5.2 we obtain the result of Thm. 5.16 for $\alpha \geq 1$.

5.3.4 β-Aperiodic Palindromes

We briefly explain the main differences and similarities needed to understand the relationship between gapped repeats and palindromes. Let (U_L, U_R) be a maximal α-gapped repeat (resp. α-gapped palindrome). If its right arm U_R has a periodic prefix S_R generated by a run, then its left arm has a periodic prefix S_L (resp. periodic suffix S_L) generated by a run of the same period. Since (U_L, U_R)

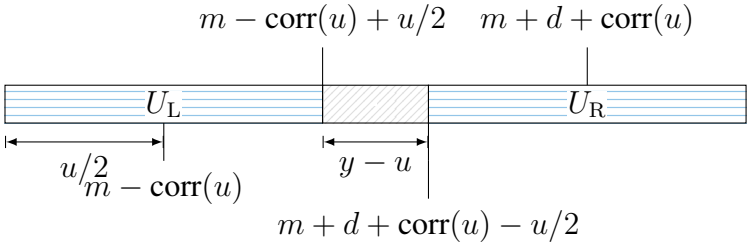


Fig. 5.23: A gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ with $u = |U_\mathrm{L}|$ mapped to the point (m, d).

is maximal, both runs have to obey constraints that are similar in both cases, considering whether $(U_\mathrm{L}, U_\mathrm{R})$ is a gapped repeat or a gapped palindrome (this is reflected by the fact that large parts for proving the statements of Lemma 5.7($\mathcal{R}$) and Lemma 5.7($\mathcal{P}$) are identical). Like with aperiodic gapped repeats, we can apply the point analysis to the aperiodic α-gapped palindromes, too. To this end, we map a maximal α-gapped β-aperiodic palindrome $(U_\mathrm{L}, U_\mathrm{R})$ of a string of length n to the point

$$(m, d) \coloneqq \xi_\mathcal{P}(U_\mathrm{L}, U_\mathrm{R}) \coloneqq (\lceil(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2\rceil\,, \\ \lfloor(\mathsf{b}(U_\mathrm{R}) + \mathsf{e}(U_\mathrm{R}))/2\rfloor - \lceil(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2\rceil),$$

where $\xi_\mathcal{P}$ denotes this mapping. Let $\xi_\mathcal{P}(\mathcal{P}_\alpha(T)) = \{\xi_\mathcal{P}(U_\mathrm{L}, U_\mathrm{R}) \mid (U_\mathrm{L}, U_\mathrm{R})$ is a maximal α-gapped β-aperiodic palindrome $\} \subset \mathcal{C}_n$ be the image of $\xi_\mathcal{P}$. The first coordinate m is the (integer) position nearest to the mid-point $(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2$ (tie-breaking to the *right*) of the left arm, and $m + d$ is the position nearest to the mid-point $(\mathsf{b}(U_\mathrm{R}) + \mathsf{e}(U_\mathrm{R}))/2$ (tie-breaking to the *left*) of the right arm (in particular, $T[m] = T[m + d]$). The mapping $\xi_\mathcal{P}$ is injective because we can retrieve the pair of segments $(U_\mathrm{L}, U_\mathrm{R})$ by computing the maximal inward and outward matches at the positions m and $m + d$. Since m and d are positive integers with $m + d \le n$, we conclude that $(m, d) \in \mathcal{C}_n$. For convenience, we give an alternative definition of (m, d) using the function[3] $\mathsf{corr}(i) \coloneqq (i + 1 \bmod 2)/2$ such that $\mathsf{corr}(i) = 0$ if i is odd, and $\mathsf{corr}(i) = 1/2$ if i is even. With $\mathsf{corr}(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}) + 1) = \mathsf{corr}(2\mathsf{b}(U_\mathrm{L}) + |U_\mathrm{L}|) = \mathsf{corr}(|U_\mathrm{L}|)$ we get $(m, d) = ((\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2 + \mathsf{corr}(|U_\mathrm{L}|), q - 2\mathsf{corr}(|U_\mathrm{L}|))$, where $q \coloneqq \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$ is the arm-period of $(U_\mathrm{L}, U_\mathrm{R})$ (see also Fig. 5.23).

Fact 5.17. Given a maximal gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ with $u \coloneqq |U_\mathrm{L}|$ and $(m, d) \coloneqq \xi_\mathcal{P}(U_\mathrm{L}, U_\mathrm{R})$, it holds that

(a) $|u/2 - \mathsf{corr}(u)| \in \{1/2, 3/2, 5/2, \ldots\}$,

(b) $\mathsf{b}(U_\mathrm{L}) = m - \mathsf{corr}(u) - u/2 + 1/2$,

(c) $\mathsf{e}(U_\mathrm{L}) = m - \mathsf{corr}(u) + u/2 - 1/2$, and

(d) $\mathsf{b}(U_\mathrm{R}) = m + d + \mathsf{corr}(u) - u/2 + 1/2$.

[3] The function is called corr since it calculates a *correction* term that adds with $(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2$ up to the mid-point m of the left arm.

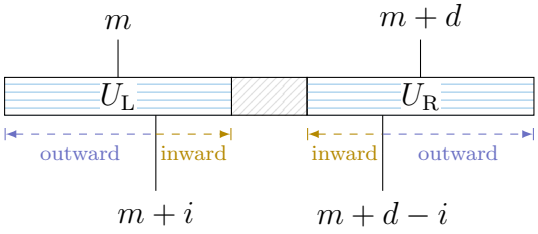


Fig. 5.24: Setting of Lemma 5.18. A gapped palindrome $(U_{\mathrm{L}}, U_{\mathrm{R}})$ is mapped to the point (m, d). It can be restored with LCE and LCS queries at the positions $m+i$ and $d-2i$ for every integer i with $-\lfloor u/2 \rfloor - 1 \leq i \leq -1$ or $1 \leq i \leq \lceil u/2 \rceil$, where $u \coloneqq |U_{\mathrm{L}}| = |U_{\mathrm{R}}|$.

(e) If $d \leq 2$, then $(U_{\mathrm{L}}, U_{\mathrm{R}})$ is a maximal *ordinary* palindrome.

Lemma 5.18. Given a maximal α-gapped β-aperiodic palindrome $(U_{\mathrm{L}}, U_{\mathrm{R}})$ with $u \coloneqq |U_{\mathrm{L}}|$ and $(m, d) = \xi_{\mathcal{P}}(U_{\mathrm{L}}, U_{\mathrm{R}})$, $(m+i, d-2i) \notin \xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$ for every integer i with $-\lfloor u/2 \rfloor - 1 \leq i \leq -1$ or $1 \leq i \leq \lceil u/2 \rceil$.

Proof. For every integer i with $-\lfloor u/2 \rfloor \leq i \leq -1$ or $1 \leq i \leq \lceil u/2 \rceil - 1$ (excluding $-\lfloor u/2 \rfloor - 1$ and $\lceil u/2 \rceil$ as stated in the claim), the maximal inward and outward matches at the positions $m+i$ and $m+d-i$ yields $(U_{\mathrm{L}}, U_{\mathrm{R}})$, and thus, $(m+i, d-2i)$ cannot be in $\xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$ due to the injectivity of $\xi_{\mathcal{P}}$ (cf. Fig. 5.24). If i is $-\lfloor u/2 \rfloor - 1$ or $\lceil u/2 \rceil$, the point $(m+i, d-2i) \in \mathbb{Z}^2$ is not in $\xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$ because the pair of positions $m+i$ and $m+d-i$ is where the inward or outward match from the positions m and $m+i$ fails. □

Due to Lemma 5.18, each point $(m, d) \in \xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$ has at least one distinct point that is not in the image of $\xi_{\mathcal{P}}$. For instance, we count the point $(m+1, d-2) \in \mathcal{C}_n \setminus \xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$ ($d \geq 3$ according to Fact 5.17(e)) for each $(m, d) \in \xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$, and each counted point is counted only once. With this insight, we can prove the next corollary in exactly the same way as Lemma 5.13.

Corollary 5.19. Let γ be a real number with $\gamma \in (0, 1]$. A set of points $C \subseteq \xi_{\mathcal{P}}(\mathcal{P}_\alpha(T))$ such that no two distinct points in C γ-cover the same point obeys the inequality $|C| < n(\pi^2/6 - 1/2)/\gamma$.

Proof. With the same definition of E as in the proof of Lemma 5.13, it is left to show that the sum of the weights of all points in E is at most $n/(2\gamma)$.

Unlike the proof of Lemma 5.13, we can take a shortcut with the following observation: Only the highest points in E can be γ-covered by a point from $C \setminus E$.[4] To see this, let $(x, y) \in E$ be a point with $y < 1/\gamma - 1$. Assume that $(\hat{x}, \hat{y})$ γ-covers (x, y), then $\hat{y} - \gamma\hat{y} \leq y < 1/\gamma - 1$, or equivalently $\hat{y} < 1/\gamma$. This means that $(x, y) = (\hat{x}, \hat{y})$.

We conclude that every point $(x, y) \in E$ with $\mathsf{weight}(x, y) = 1$ belongs to C, and therefore (a) $(x+1, y-2) \notin C$ according to Lemma 5.18, and (b) $\mathsf{weight}(x+1, y-2) = 0$ according to the above observation. Hence, both points (x, y)

[4] This holds also in the case of maximal α-gapped repeats in the proof of Lemma 5.13. However, this trick does not lead to anything useful there.

and $(x+1, y-2)$ have a total weight of 1 (remember that $\mathsf{weight}(n, y-2) = 0$ in any case, cf. proof of Lemma 5.13). For $y \leq 2$, there is no aperiodic α-gapped palindrome that can be mapped to a point (x, y) according to Fact 5.17(e).

Although a highest point (x, y) (with $1/\gamma - 1 \leq y$) can be γ-covered by a point in $C \setminus E$, one of its neighbors $(x-1, y)$ or $(x+1, y)$ has to be γ-covered by the same point, such that the sum of the weights of both points is at most $1/2$. The total weight of all points in E is therefore at most $(1/2)\,|E| \leq n/(2\gamma)$. □

Similar to Lemma 5.14, we can apply the result of Cor. 5.19 to the set $\overline{\mathcal{P}_\alpha^\beta}(T)$ by restricting the domain of β:

Lemma 5.20. Let T be a string, and α and β two real numbers with $\alpha > 1$ and $6/7 \leq \beta < 1$. The points mapped by two different maximal gapped palindromes in $\overline{\mathcal{P}_\alpha^\beta}(T)$ cannot $\frac{1-\beta}{\alpha}$-cover the same point.

Proof. Let $(U_\mathrm{L}, U_\mathrm{R})$ and $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ be two different maximal α-gapped palindromes in $\overline{\mathcal{P}_\alpha^\beta}(T)$. Set $u := |U_\mathrm{L}| = |U_\mathrm{R}|$ and $\overline{u} := \left|\overline{U_\mathrm{L}}\right| = \left|\overline{U_\mathrm{R}}\right|$. Let (m, d) and $(\overline{m}, \overline{d})$ be the points mapped from $(U_\mathrm{L}, U_\mathrm{R})$ and $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$, respectively. Assume, for the sake of contradiction, that both points $\frac{1-\beta}{\alpha}$-cover the same point (x, y).

Let $\zeta := |m - \overline{m}|$, and let $S_\mathrm{L} := U_\mathrm{L} \cap \overline{U_\mathrm{L}}$ be the overlap of U_L and $\overline{U_\mathrm{L}}$. Let $s := |S_\mathrm{L}|$, and let S_R (resp. $\overline{S_\mathrm{R}}$) be the reverse copy of S_L based on $(U_\mathrm{L}, U_\mathrm{R})$ (resp. $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$), i.e., $S_\mathrm{L} = S_\mathrm{R}{}^\mathsf{T} = \overline{S_\mathrm{R}}{}^\mathsf{T}$ with $\mathsf{b}(S_\mathrm{R}) = \mathsf{b}(U_\mathrm{R}) + \mathsf{e}(U_\mathrm{L}) - \mathsf{e}(S_\mathrm{L})$ and $\mathsf{b}(\overline{S_\mathrm{R}}) = \mathsf{b}(\overline{U_\mathrm{R}}) + \mathsf{e}(\overline{U_\mathrm{L}}) - \mathsf{e}(S_\mathrm{L})$.

Sub-Claim. The overlap S_L is not empty, and $\mathsf{b}(S_\mathrm{R}) \neq \mathsf{b}(\overline{S_\mathrm{R}})$.

Sub-Proof. First we show that S_L is not empty. If $m = \overline{m}$, it is clear that S_L contains $T[m]$. Without loss of generality, assume that $m < \overline{m}$ for this sub-proof (otherwise exchange $(U_\mathrm{L}, U_\mathrm{R})$ with $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$). By combining (a) the $(1-\beta)/\alpha$-cover property with (b) the fact that $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ is α-gapped and (c) the constraint $6/7 \leq \beta < 1$, we obtain $\overline{m} - \overline{u}/2 \leq_{(\mathrm{c})} \overline{m} - (1-\beta)\overline{u} \leq_{(\mathrm{b})} \overline{m} - \overline{d}(1-\beta)/\alpha \leq_{(\mathrm{a})} x \leq_{(\mathrm{a})} m < \overline{m}$. This long inequality says that the text position m is contained in $\overline{U_\mathrm{L}}$, which implies that S_L is not empty. If S_R and $\overline{S_\mathrm{R}}$ start at the same position, then expanding the arms S_L and $S_\mathrm{R}(\equiv \overline{S_\mathrm{R}})$ to the left and right yields the arms $U_\mathrm{L} \equiv \overline{U_\mathrm{L}}$ and $U_\mathrm{R} \equiv \overline{U_\mathrm{R}}$, which implies that $(U_\mathrm{L}, U_\mathrm{R})$ and $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}})$ are the same gapped repeat, a contradiction. ■

Without loss of generality let $d \leq \overline{d}$. With the $(1-\beta)/\alpha$-cover property we obtain

$$\overline{d} - \frac{\overline{d}(1-\beta)}{\alpha} \leq y \leq d \leq \overline{d}. \tag{5.8}$$

The difference $\delta := \overline{d} - d \geq 0$ can be estimated by

$$\delta \leq \overline{d}(1-\beta)/\alpha \leq \overline{u}(1-\beta). \tag{5.9}$$

Fig. 5.25: Sub-Case 1a in the proof of Lemma 5.20 with $m \leq \overline{m}$ and $\mathsf{b}(U_L) \leq \mathsf{b}(\overline{U_L}) \leq \mathsf{e}(\overline{U_L}) \leq \mathsf{e}(U_L)$.

Equation (5.8) can also be used to lower bound u in terms of $\overline{d}$ due to the fact that (U_L, U_R) is α-gapped:

$$u \geq d/\alpha \geq \frac{\overline{d}}{\alpha}(1 - \frac{1-\beta}{\alpha}) \geq \overline{d}\beta/\alpha. \tag{5.10}$$

Outline. In the following we conduct a thorough case analysis. In each case we show the contradiction that (U_L, U_R) or $(\overline{U_L}, \overline{U_R})$ are β-periodic. We prove each case in a similar way: We first show that the intersection of S_R and $\overline{S_R}$ is large enough such that it induces a repetition on $S_R \cup \overline{S_R}$. Subsequently, we find a run covering $S_R \cup \overline{S_R}$, and another run covering S_L. However, since S_L is the suffix of U_L (resp. $\overline{U_L}$), we can conclude that (U_L, U_R) (resp. $(\overline{U_L}, \overline{U_R})$) is β-periodic.

Before starting with the case analysis, we introduce a general property of the starting positions $\mathsf{b}(S_R)$ and $\mathsf{b}(\overline{S_R})$ needed for the analysis. Adding up the equalities of Fact 5.17(c) and (d) gives $\mathsf{b}(U_R) + \mathsf{e}(U_L) = 2m + d$. With that we obtain $\mathsf{b}(S_R) = \mathsf{b}(U_R) + \mathsf{e}(U_L) - \mathsf{e}(S_L) = 2m + d - \mathsf{e}(S_L)$. Hence, the distance between the starting positions of S_R and $\overline{S_R}$ is given by

$$\left|\mathsf{b}(S_R) - \mathsf{b}(\overline{S_R})\right| = \begin{cases} 2\zeta + \delta & \text{if } m \leq \overline{m}, \\ 2\zeta - \delta & \text{if } m > \overline{m} \text{ and } \mathsf{b}(S_R) > \mathsf{b}(\overline{S_R}), \text{ or} \\ \delta - 2\zeta & \text{if } m > \overline{m} \text{ and } \mathsf{b}(S_R) < \mathsf{b}(\overline{S_R}). \end{cases} \tag{5.11}$$

Case 1: $m \leq \overline{m}$. Since $\overline{m} - \overline{d}(1-\beta)/\alpha \leq x \leq m \leq \overline{m}$ (due to the $(1-\beta)/\alpha$-cover property),

$$\zeta = \overline{m} - m \leq \overline{d}(1-\beta)/\alpha \leq \overline{u}(1-\beta), \tag{5.12}$$

because $(\overline{U_L}, \overline{U_R})$ is α-gapped. Due to Eq. (5.11), the starting positions of both right copies $\overline{S_R}$ and S_R differ by $\mathsf{b}(\overline{S_R}) - \mathsf{b}(S_R) = 2\zeta + \delta > 0$. By Eqs. (5.9) and (5.12), we get

$$2\zeta + \delta \leq 3\overline{d}(1-\beta)/\alpha \leq 3\overline{u}(1-\beta). \tag{5.13}$$

Depending on the relationships between $\mathsf{b}(U_L)$ and $\mathsf{b}(\overline{U_L})$, and between $\mathsf{e}(U_L)$ and $\mathsf{e}(\overline{U_L})$, we split the case in four sub-cases. However, one of the four sub-cases with $\mathsf{b}(\overline{U_L}) < \mathsf{b}(U_L)$ and $\mathsf{e}(\overline{U_L}) < \mathsf{e}(U_L)$ already leads to a contradiction (without proving that one left arm has a periodic suffix): Assume that both

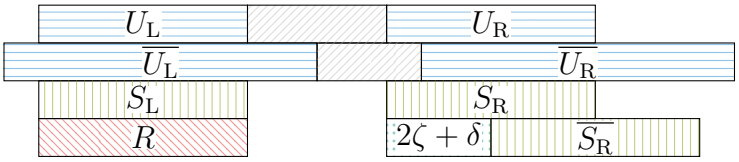


Fig. 5.26: Sub-Case 1b in the proof of Lemma 5.20 with $m \le \overline{m}$ and $\mathsf{b}(\overline{U_L}) \le \mathsf{b}(U_L) \le \mathsf{e}(U_L) \le \mathsf{e}(\overline{U_L})$.

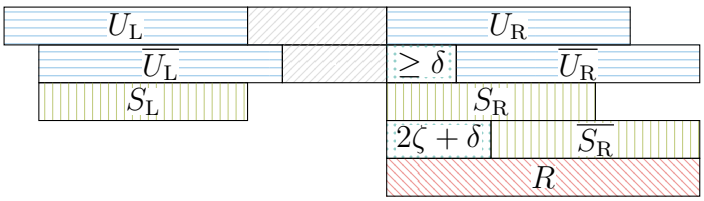


Fig. 5.27: Sub-Case 1c in the proof of Lemma 5.20 with $m \le \overline{m}$ and $\mathsf{b}(U_L) < \mathsf{b}(\overline{U_L}) \le \mathsf{e}(U_L) < \mathsf{e}(\overline{U_L})$. The second inequality holds because the overlap S_L cannot be empty due to the sub-claim.

inequalities $\mathsf{b}(\overline{U_L}) < \mathsf{b}(U_L)$ and $\mathsf{e}(\overline{U_R}) < \mathsf{e}(U_R)$ hold for the sake of contradiction. Under these assumptions, with Fact 5.17(b) it must hold that $\mathsf{b}(\overline{U_L}) + 1/2 = \overline{m} - \mathsf{corr}(\overline{u}) - \overline{u}/2 + 1 \le m - \mathsf{corr}(u) - u/2 = \mathsf{b}(U_L) - 1/2$ and $\mathsf{e}(\overline{U_R}) + 1/2 = \overline{m} - \mathsf{corr}(\overline{u}) + \overline{u}/2 + 1 \le m - \mathsf{corr}(u) + u/2 = \mathsf{e}(U_R) - 1/2$. Adding the left sides and the right sides of both inequalities gives $\overline{m} - m \le \mathsf{corr}(\overline{u}) - \mathsf{corr}(u) - 1 < 0$, which contradicts that $\overline{m} - m \ge 0$.

Thus it is enough to consider the following three sub-cases 1a, 1b, and 1c.

Sub-Case 1a: $S_L \equiv \overline{U_L}$, see Fig. 5.25. Since $\overline{u}/(2\zeta+\delta) \ge \overline{u}/(3\overline{u}(1-\beta)) \ge 7/3 > 2$ holds (due to Eq. (5.13)) for $6/7 \le \beta < 1$, we conclude that $S_R = \overline{U_R}$ is periodic, which means that $(\overline{U_L}, \overline{U_R}) \in \mathcal{P}_\alpha^\beta(T)$, a contradiction.

Sub-Case 1b: $S_L \equiv U_L$, see Fig. 5.26. Recall that $u = s \ge \overline{d}\beta/\alpha$ by Eq. (5.10). It follows from Eq. (5.13) and $6/7 \le \beta < 1$ that $s/(2\zeta+\delta) \ge \overline{d}\alpha\beta/(3\overline{d}\alpha(1-\beta)) = \beta/(3(1-\beta)) \ge 2$. Hence $S_R \equiv U_R$ is periodic, which means that $(U_L, U_R) \in \mathcal{P}_\alpha^\beta(T)$, a contradiction.

Sub-Case 1c: $\mathsf{b}(U_L) < \mathsf{b}(\overline{U_L})$ and $\mathsf{e}(U_L) < \mathsf{e}(\overline{U_L})$, see Fig. 5.27. Since S_L is a suffix of U_L and a prefix of $\overline{U_L}$, the reverse copies S_R and $\overline{S_R}$ are a prefix of U_R and a suffix of $\overline{U_R}$, respectively. We have $1 \le \mathsf{b}(\overline{U_L}) - \mathsf{b}(U_L) = \overline{m} - \overline{u}/2 - \mathsf{corr}(\overline{u}) - (m - u/2 - \mathsf{corr}(u))$. A simple reshaping leads to $\overline{m} - \overline{u}/2 - (m - u/2) \ge 1 + \mathsf{corr}(\overline{u}) - \mathsf{corr}(u)$. This inequality yields $\mathsf{b}(\overline{U_R}) - \mathsf{b}(U_R) = \overline{m} + \overline{d} - \overline{u}/2 + \mathsf{corr}(\overline{u}) - (m + d - u/2 + \mathsf{corr}(u)) \ge \delta + 1 + 2(\mathsf{corr}(\overline{u}) - \mathsf{corr}(u)) \ge \delta \ge 0$. This means that $\mathsf{b}(S_R) = \mathsf{b}(U_R) \le \mathsf{b}(\overline{U_R}) \le \mathsf{b}(\overline{S_R}) \le \mathsf{e}(\overline{S_R}) = \mathsf{e}(\overline{U_R})$. With $\mathsf{b}(S_R) \le \mathsf{b}(\overline{U_R}) = \mathsf{e}(\overline{S_R}) - \overline{u} + 1 = \mathsf{b}(\overline{S_R}) + s - \overline{u}$, it follows that $s \ge \overline{u} - (2\zeta+\delta) > 2\zeta + \delta$ because $\overline{u}/(2\zeta+\delta) \ge \overline{u}/(3\overline{u}(1-\beta)) \ge 7/3 > 2$ holds for $6/7 \le \beta < 1$. Since $\mathsf{b}(\overline{S_R}) - \mathsf{b}(S_R) = \mathsf{e}(\overline{S_R}) - \mathsf{e}(S_R) = 2\zeta + \delta$, $S_R \cap \overline{S_R} \ne \emptyset$. Putting everything together, we have that $\overline{U_R} \subset S_R \cup \overline{S_R}$, and that $\overline{U_R}$ is periodic with a period of at most $2\zeta + \delta$, a contradiction.

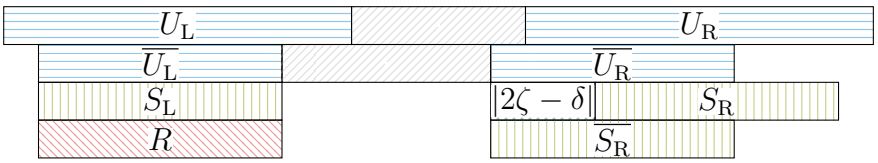


Fig. 5.28: Sub-Case 2a in the proof of Lemma 5.20 with $m > \overline{m}$ and $\mathsf{b}(U_\mathrm{L}) \leq \mathsf{b}(\overline{U_\mathrm{L}}) \leq \mathsf{e}(\overline{U_\mathrm{L}}) \leq \mathsf{e}(U_\mathrm{L})$.

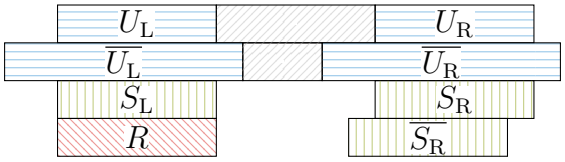


Fig. 5.29: Sub-Case 2b in the proof of Lemma 5.20 with $m > \overline{m}$ and $\mathsf{b}(\overline{U_\mathrm{L}}) \leq \mathsf{b}(U_\mathrm{L}) \leq \mathsf{e}(U_\mathrm{L}) \leq \mathsf{e}(\overline{U_\mathrm{L}})$.

Case 2: $m > \overline{m}$. Since $m - d(1-\beta)/\alpha \leq x \leq \overline{m} < m$,

$$\zeta = m - \overline{m} \leq d(1-\beta)/\alpha \leq \overline{d}(1-\beta)/\alpha \leq \overline{u}(1-\beta). \tag{5.14}$$

According to Eq. (5.11), the starting positions of both right copies differ by $\left|\mathsf{b}(S_\mathrm{R}) - \mathsf{b}(\overline{S_\mathrm{R}})\right| = |2\zeta - \delta|$. Equation (5.11) with Eqs. (5.9) and (5.14) yields

$$|2\zeta - \delta| \leq \begin{cases} 2\zeta \leq 2\overline{d}(1-\beta)/\alpha \leq 2\overline{u}(1-\beta) & \text{if } \mathsf{b}(S_\mathrm{R}) > \mathsf{b}(\overline{S_\mathrm{R}}), \text{ or} \\ \delta \leq \overline{d}(1-\beta)/\alpha \leq \overline{u}(1-\beta) & \text{if } \mathsf{b}(S_\mathrm{R}) < \mathsf{b}(\overline{S_\mathrm{R}}). \end{cases} \tag{5.15}$$

Like Case 1, we split the case into sub-cases depending on the relation of the starting and of the ending positions of the left arms. The sub-case with $\mathsf{b}(U_\mathrm{L}) < \mathsf{b}(\overline{U_\mathrm{L}})$ and $\mathsf{e}(U_\mathrm{L}) < \mathsf{e}(\overline{U_\mathrm{L}})$ already leads to a contradiction, which can be seen by an argument that is similar to the one used in Case 1 due to symmetry.
Sub-Case 2a: $S_\mathrm{L} \equiv \overline{U_\mathrm{L}}$, see Fig. 5.28. Since $s/|2\zeta - \delta| \geq \overline{u}/(2\overline{u}(1-\beta)) = 1/(2(1-\beta)) \geq 7/2 > 2$ holds (due to Eq. (5.15)) for $6/7 \leq \beta < 1$, the distance between $\mathsf{b}(S_\mathrm{R})$ and $\mathsf{b}(\overline{S_\mathrm{R}})$ is small enough such that $S_\mathrm{R} = \overline{U_\mathrm{R}}$ is periodic, which means that $(\overline{U_\mathrm{L}}, \overline{U_\mathrm{R}}) \in \mathcal{P}_\alpha^\beta(T)$, a contradiction.
Sub-Case 2b: $S_\mathrm{L} \equiv U_\mathrm{L}$, see Fig. 5.29. Recall that $u = s \geq \overline{d}\beta/\alpha$ by Eq. (5.10). It follows from $6/7 \leq \beta < 1$ and Eq. (5.15) that $s/|2\zeta - \delta| \geq \overline{d}\alpha\beta/(2\overline{d}\alpha(1-\beta)) = \beta/(2(1-\beta)) \geq 3 > 2$. Hence $S_\mathrm{R} \equiv U_\mathrm{R}$ is periodic, which means that $(U_\mathrm{L}, U_\mathrm{R}) \in \mathcal{P}_\alpha^\beta(T)$, a contradiction.
Sub-Case 2c: $\mathsf{b}(U_\mathrm{L}) > \mathsf{b}(\overline{U_\mathrm{L}})$ and $\mathsf{e}(U_\mathrm{L}) > \mathsf{e}(\overline{U_\mathrm{L}})$. Since S_L is a prefix of U_L and a suffix of $\overline{U_\mathrm{L}}$, the reverse copies S_R and $\overline{S_\mathrm{R}}$ are a suffix of U_R and a prefix of $\overline{U_\mathrm{R}}$, respectively.
Subsub-Case 2c-i: $\mathsf{b}(U_\mathrm{R}) \geq \mathsf{b}(\overline{U_\mathrm{R}})$, see Fig. 5.30. Recall that $u \geq \overline{d}\beta/\alpha$ by Eq. (5.10). It follows from $6/7 \leq \beta < 1$ and Eq. (5.15) that $u/|2\zeta - \delta| \geq \overline{d}\alpha\beta/(2\overline{d}\alpha(1-\beta)) = \beta/(2(1-\beta)) \geq 3 > 2$. With $\mathsf{b}(U_\mathrm{R}) \geq \mathsf{b}(\overline{U_\mathrm{R}})$, this case is symmetric to Sub-Case 1c, leading to the result that $U_\mathrm{R} \subset S_\mathrm{R} \cup \overline{S_\mathrm{R}}$, and that U_R is periodic with a period of at most $|2\zeta - \delta|$, a contradiction.
Subsub-Case 2c-ii: $\mathsf{b}(U_\mathrm{R}) < \mathsf{b}(\overline{U_\mathrm{R}})$, see Fig. 5.31. It follows from $\mathsf{b}(U_\mathrm{R}) < \mathsf{b}(\overline{U_\mathrm{R}})$ that $\mathsf{b}(U_\mathrm{R}) - \mathsf{b}(\overline{U_\mathrm{R}}) = m + d - u/2 + \mathsf{corr}(u) - (\overline{m} + \overline{d} - \overline{u}/2 + \mathsf{corr}(\overline{u})) =$

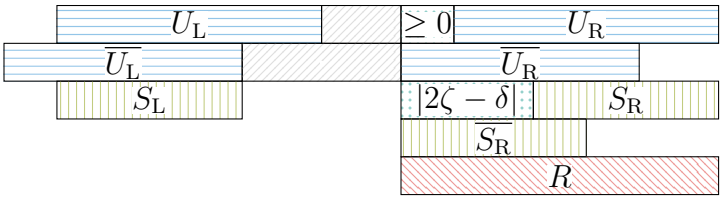


Fig. 5.30: Subsub-Case 2c-i in the proof of Lemma 5.20 with $m > \overline{m}$, $\mathsf{b}(\overline{U_L}) < \mathsf{b}(U_L) \leq \mathsf{e}(\overline{U_L}) < \mathsf{e}(U_L)$ and $\mathsf{b}(\overline{U_R}) \leq \mathsf{b}(U_R)$.

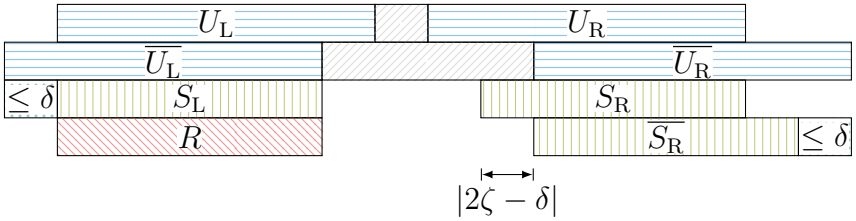


Fig. 5.31: Subsub-Case 2c-ii in the proof of Lemma 5.20 with $m > \overline{m}$, $\mathsf{b}(\overline{U_L}) < \mathsf{b}(U_L) \leq \mathsf{e}(\overline{U_L}) < \mathsf{e}(U_L)$ and $\mathsf{b}(U_R) < \mathsf{b}(\overline{U_R})$.

$\mathsf{b}(U_L) - \mathsf{b}(\overline{U_L}) - \delta + 2(\mathsf{corr}(u) - \mathsf{corr}(\overline{u})) \leq -1$, which leads to $\mathsf{b}(U_L) - \mathsf{b}(\overline{U_L}) \leq \delta + 2(\mathsf{corr}(\overline{u}) - \mathsf{corr}(u)) - 1 \leq \delta$. Combining this inequality with Eq. (5.9) gives $s = \overline{u} - (\mathsf{b}(U_L) - \mathsf{b}(\overline{U_L})) \geq \overline{u} - \delta \geq \beta\overline{u}$. With Eq. (5.15) this yields $s/|2\zeta - \delta| \geq \beta\overline{u}/(2\overline{u}(1-\beta)) = \beta/(2(1-\beta)) \geq 3 > 2$ under the presumption that $6/7 \leq \beta < 1$. Putting everything together, $\overline{U_L}$ has a periodic suffix of length $\beta\overline{u}$, and that $(\overline{U_L}, \overline{U_R}) \in \mathcal{P}_\alpha^\beta(T)$, a contradiction. □

Combining the results of Cor. 5.19 and Lemma 5.20 immediately gives the following corollary:

Corollary 5.21. Given two real numbers α and β with $\alpha > 1$ and $6/7 \leq \beta < 1$, and a string T of length n, the number of all maximal α-gapped β-aperiodic palindromes is bounded by the inequality $\left|\overline{\mathcal{P}_\alpha^\beta}(T)\right| < \alpha n(\pi^2/6 - 1/2)/(1-\beta)$.

Theorem 5.22. Given a real number α with $\alpha > 1$, and a string T of length n, the number of all maximal α-gapped palindromes $|\mathcal{P}_\alpha(T)|$ less than $7(\pi^2/6 + 1/2)\alpha n - 3n - 1$.

Proof. Combining the results of Lemma 5.7 and Cor. 5.21 yields

$$\begin{aligned}|\mathcal{P}_\alpha(T)| &= \underbrace{2n-1}_{\text{max. palindromes}} + \underbrace{\left|\mathcal{P}_\alpha^\beta(T)\right|}_{\beta\text{-periodic}} + \underbrace{\left|\overline{\mathcal{P}_\alpha^\beta}(T)\right|}_{\beta\text{-aperiodic}} \\ &< \underbrace{2n-1}_{\text{Lemma 2.3}} + \underbrace{2(\alpha-1)\frac{\mathcal{E}(T)}{\beta} + 2n}_{\text{Lemma 5.7}} + \underbrace{\left(\frac{\pi^2}{6} - \frac{1}{2}\right)\frac{\alpha n}{1-\beta}}_{\text{Cor. 5.21}}\end{aligned}$$

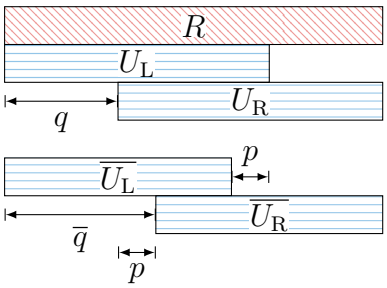


Fig. 5.32: Two gapped repeats (U_L, U_R) and $(\overline{U_L}, \overline{U_R})$ with overlapping arms. Both gapped repeats are within a run R. They are maximal if their arms border the run R. Each such maximal gapped repeat with overlapping arms has an arm-period (q or $\overline{q}$ in the figure) that is a multiple of R's smallest period p.

for every $6/7 \leq \beta < 1$. Applying Lemma 2.1, the term on the right side is upper bounded by $4n - 1 + 2(\alpha - 1)(3n/\beta) + (\pi^2/6 - 1/2)\alpha n/(1 - \beta)$. This number is minimal when $\beta = 6/7$, yielding the bound $4n-1+7n(\alpha-1)+7(\pi^2/6-1/2)\alpha n = 7(\pi^2/6 + 1/2)\alpha n - 3n - 1$. □

5.4 Computing All Maximal α-Gapped Repeats

We split the set of all maximal α-gapped repeats into four subsets. For each subset, we devise an algorithm that computes this subset in $\mathcal{O}(\alpha n)$ time. We assign a maximal α-gapped repeat/palindrome to a subset depending on the condition whether it has

Set 1: arms consisting of one character,

Set 2: arms that overlap,

Set 3: non-overlapping arms with a length between 2 and $\gamma \lg n$ characters, for an integer constant γ with $\gamma \geq 4$, or

Set 4: non-overlapping arms longer than $\gamma \lg n$ characters.

As a starter, we can find Set 1 very easily in our target time of $\mathcal{O}(\alpha n)$:

Lemma 5.23. We can compute all maximal α-gapped repeats/palindromes with an arm of one character in a string T of length n in $\mathcal{O}(\alpha n)$ time.

Proof. For each position i with $1 \leq i \leq n$, we check whether the characters $T[i]$ and $T[i + j]$ are equal for each position j with $1 \leq j \leq \alpha$. If both characters are equal, they form an α-gapped repeat. In particular, they are the arms of a *maximal* α-gapped repeat if we can prolong $T[i]$ and $T[i + j]$ neither to the left nor to the right (check $T[i - 1] \neq T[i + j - 1]$ and $T[i + 1] \neq T[i + j + 1]$). □

5.4.1 Overlapping Arms

We provide an $\mathcal{O}(n)$ time algorithm that finds all maximal α-gapped repeats/-palindromes (U_L, U_R) of Set 2, i.e., with with $\mathsf{e}(U_L) \geq \mathsf{b}(U_R)$. When studying

α-gapped repeats/palindromes of Set 2, we can neglect the parameter α, because a gapped repeat/palindrome $(U_\mathrm{L}, U_\mathrm{R})$ whose arms overlap obeys the inequality $\mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L}) < |U_\mathrm{L}| \leq \alpha\,|U_\mathrm{L}|$ for every $\alpha \geq 1$. For a gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ with $\mathsf{e}(U_\mathrm{L}) \geq \mathsf{b}(U_\mathrm{R})$, we already know that either $(U_\mathrm{L}, U_\mathrm{R})$ is not maximal, or $U_\mathrm{L} \equiv U_\mathrm{R}$. Remembering that a maximal gapped palindrome with an overlap is equal to a maximal ordinary palindrome, we are already done because we can compute all maximal ordinary palindromes in $\mathcal{O}(n)$ time according to Lemma 2.4. It remains to focus on the maximal gapped repeats of Set 2:

Lemma 5.24. We can compute all maximal gapped repeats with overlapping arms in $\mathcal{O}(n)$ time.

Proof. Given a maximal gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ with arm-period $q \coloneqq \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L}) < |U_\mathrm{L}|$, it induces a square with $T[\mathsf{b}(U_\mathrm{L}) .. \mathsf{b}(U_\mathrm{L})+q-1] = T[\mathsf{b}(U_\mathrm{R}) .. \mathsf{b}(U_\mathrm{R})+q-1]$. The square induces a run R whose smallest period p divides q (also observed in [66, Conclusions]). Both arms U_L and U_R are contained in R. Because $(U_\mathrm{L}, U_\mathrm{R})$ is maximal, $\mathsf{b}(U_\mathrm{L}) = \mathsf{b}(R)$ and $\mathsf{e}(U_\mathrm{R}) = \mathsf{e}(R)$ hold; otherwise we could extend the arms to the left or to the right, respectively. This means that the left arm U_L covers at least the segment $T[\mathsf{b}(R) .. \mathsf{b}(R) + \exp(R)p/2]$ (otherwise the arms would not overlap). Since q is a multiple of p, the number of different lengths of U_L is bounded by $\exp(R)/2$. Figure 5.32 illustrates two maximal gapped repeats with overlapping arms within the same run.

Our idea is to probe at the borders of each run R for all possible values of q to find all gapped repeats whose arms (a) overlap and (b) are contained in R. To find these borders, we first compute all runs in linear time with Lemma 2.2. Then we linearly scan over all runs from left to right in text order. Having $\mathsf{LCE}^{\leftrightarrow}$ in the representation of Cor. 2.12, we spend $\mathcal{O}(\exp(R))$ time on each run R, summing up to $\mathcal{O}(n)$ time due to Lemma 2.1. Since a gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ with overlapping arms is uniquely defined by its arm-period and the borders of the run containing U_L and U_R, we can report each such gapped repeat exactly once. □

5.4.2 Support Data Structures

In the following, we focus on the gapped repeats of Sets 3 and 4. The main tool of our algorithms dealing with Sets 3 and 4 is $\mathsf{LCE}^{\leftrightarrow}$ for finding maximal equal segments of a string that start or end at particular positions. We build $\mathsf{LCE}^{\leftrightarrow}$ in the representation of Cor. 2.12 using $\mathcal{O}(n)$ words while supporting LCE and LCS queries in $t_\mathrm{LCE} = \mathcal{O}(1)$.

Our first goal is to find a representation of all occurrences of a substring Y within a string Z of length $\ell\,|Y|$ with the properties that the representation (a) can be stored in $\mathcal{O}(\ell)$ words, and (b) supports a sequential scan that retrieves an occurrence in constant time. If Y is aperiodic, two occurrences of Y cannot have

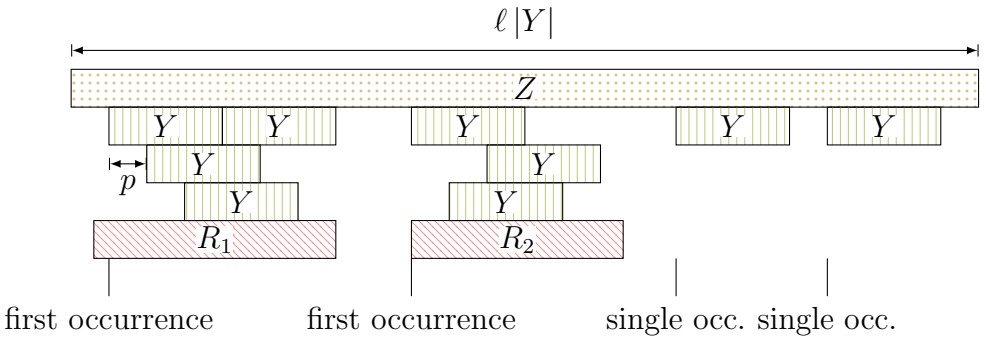


Fig. 5.33: Occurrences of Y in Z. The two rightmost segments are single occurrences. The other segments are occurrences within a run. We store the starting position of the first occurrence of Y appearing in a run, and the starting positions of the segments that are not part of a run.

an overlap with more than $Y/2$ position according to Cor. 5.6. Consequently, there can be at most 2ℓ occurrences of Y in Z. We call these occurrences *single occurrences*, and store the starting position of each occurrence in an array.

If Y is periodic with smallest period p, there can be $\mathcal{O}(\ell\,|Y|\,/p)$ occurrences of Y in Z, which can be $\Omega(\ell)$ for a sufficiently small p. That is because occurrences of Y can overlap, creating a run R with smallest period p. Instead of storing the starting positions of all occurrences of Y within R, it suffices to store only the first occurrence. The other occurrences within the run R can be computed by an arithmetic progression with common difference p. More formally (see Fig. 5.33), the segment $Z[i\,..\,i+|Y|-1]$ is

- a *single occurrence* if Y occurs neither at position $i-p$ nor at position $i+p$ in Z,
- *within a run* if Y occurs at position $i-p$ (given $i-p\geq 1$ or at position $i+p$ (given $i+p\leq |Z|$) in Z, and
- a *first occurrence* if it is within a run, but Y does not occur at position $i-p$.

According to Cor. 5.5, there are at most $\mathcal{O}(\ell)$ runs containing occurrences of Y in Z, i.e., $\mathcal{O}(\ell)$ first occurrences of Y in Z.

Corollary 5.25 ([70, Remark 1])**.** Given a substring Y of T and a segment Z of T with length $\ell\,|Y|$, the occurrences of Y in Z can be represented succinctly in $\mathcal{O}(\ell)$ words.

Proof. We only store the starting position of the single and first occurrences, and the smallest period of Y. This is sufficient, since we can reconstruct the missing information in constant time with $\mathsf{LCE}^{\leftrightarrow}$ according to Cors. 2.11 and 2.12.

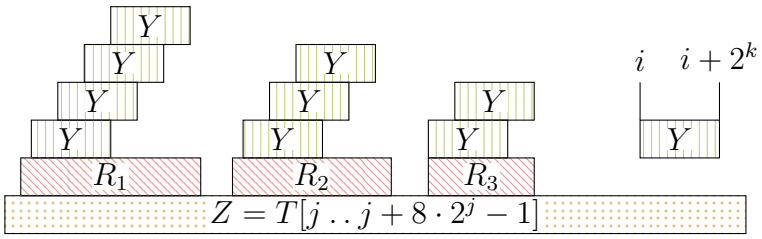


Fig. 5.34: Occurrences of the basic factor $Y = T[i \mathinner{..} i + 2^k - 1]$ in $Z = T[j \mathinner{..} j + 8 \cdot 2^k - 1]$ as described in Lemma 5.26, with $\ell := 8$. The overlapping occurrences are part of a run. The occurrences within a run R_j with $j = 1, \ldots, 3$ are represented only by the first (leftmost) occurrence of Y in R_j. In total, the representation of the occurrences of Y in Z is composed of, along with the smallest period of Y, four starting positions: three first occurrences and one single occurrence.

Having the starting position of the first occurrence of Y in a run R, we can compute all occurrences within the run R by an arithmetic progression with the common difference equal to the smallest period of Y. The number of occurrences within this run can be determined in constant time due to Cor. 2.11. □

Our approach is based on the technique of Karp et al. [150], where substrings of the length 2^k for an integer $k \geq 1$ are used to query whether two arbitrary substrings are equal. In the literature, these substrings of the length 2^k for $k \geq 1$ are called *basic factors* [59, 70, 157]. We additionally write *k-basic factor* for a basic factor of length 2^k. The algorithmic idea of Dumitran et al. [70] is to search for α-gapped repeats whose arms are basic factors, and then extend the arms to produce maximal α-gapped repeats. To find those α-gapped repeats, we need a data structure that finds the occurrences of a basic factor. Luckily, we can find the occurrences of a basic factor Y in a segment Z of length $\ell\,|Y|$ efficiently due to the following lemma:

Lemma 5.26 ([70, Lemma 1])**.** Let T be a string of length n. Given an integer $\ell \geq 2$, a basic factor Y, and a segment Z of T with $|Z| = \ell\,|Y|$, there is a data structure that reports all occurrences of Y in Z within the representation described in Cor. 5.25 in $\mathcal{O}(\lg\lg n + \ell)$ time. It can be constructed on T in $\mathcal{O}(n \lg n)$ time, taking $\mathcal{O}(n \lg n)$ words of space.

Figure 5.34 gives a visual representation of the query described in Lemma 5.26.

To accelerate the search to $\mathcal{O}(\alpha n)$ time, we devise techniques separately tailored to Sets 3 and 4. We start with Set 3. Our idea is to first spot the right arm U_{R}, and then to apply some techniques to find the left arm U_{L}: If we cover the string T with the set of segments

$$\left\{ T[m \lg n + 1 \mathinner{..} (m + \gamma + 1) \lg n] \mid 0 \leq m \leq \frac{n}{\lg n} - \gamma - 1 \right\},$$

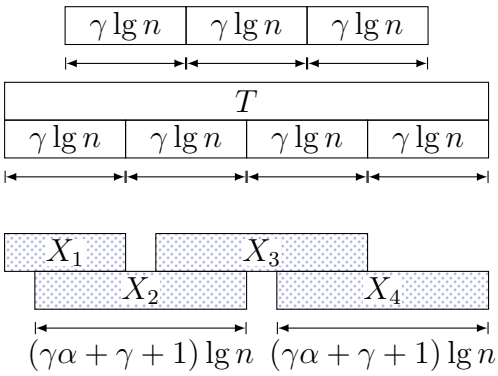


Fig. 5.35: Covering the string T with the superblocks X_m of Eq. (5.16).

then the right arm of each α-gapped repeat of Set 3 is contained in at least one of these segments. For such an α-gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$, the arm-period $\mathsf{b}(U_{\mathrm{R}}) - \mathsf{b}(U_{\mathrm{L}})$ is at most $\alpha\gamma \lg n$. By stretching every segment of the above cover to the left, the complete gapped repeat is contained in exactly one segment

(5.16)
$$X_m := \begin{cases} T[1\,..\,(m+\gamma+1)\lg n] & \text{if } (m-\gamma\alpha)\lg n + 1 \leq 0, \\ T[(m-\gamma\alpha)\lg n + 1\,..\,(m+\gamma+1)\lg n] & \text{else,} \end{cases}$$

for an integer m with $\gamma\alpha \leq m \leq \frac{n}{\lg n} - \gamma - 1$. We call each X_m a *superblock* (see Fig. 5.35). Our task is to enhance each superblock with a data structure that supports querying for all possible positions of the left arm. We show that this query can be answered efficiently under the invariant that the right arm is always contained in the last $\gamma \lg n$ characters of a superblock. The main idea is to use a bit vector marking the starting positions of a basic factor instead of relying on the data structure described in Lemma 5.26. Nevertheless, we need Lemma 5.26 for computing Set 4 (see Lemma 5.31).

Lemma 5.27 ([70, Lemma 3])**.** Let β be an integer constant with $\beta > \gamma$, and let X be a string with $|X| = \beta \lg n$. We can build a data structure on X in $\mathcal{O}(\beta \lg n)$ time that can, given a basic factor $Y = X[j2^k + 1\,..\,(j+1)2^k]$ with $j, k \geq 0$ and $j2^k + 1 > (\beta - \gamma)\lg n$, compute a bit vector of the length of X marking the beginning positions of the occurrences Y within X in $\mathcal{O}(\beta)$ time. The data structure takes $\mathcal{O}(\beta \lg n)$ words of space.

After endowing a string X with the data structure of Lemma 5.27, it is easy to find the occurrences of a basic factor Y in a small segment Z of X (see also Fig. 5.36):

Lemma 5.28 ([70, Remark 2])**.** Let Y and X be defined as in Lemma 5.27, and let Z be a segment of X with length $\ell\,|Y|$. Given the bit vector of Lemma 5.27 marking the starting positions of all occurrences of Y in X, we can represent all occurrences of Y in Z by the representation described in Cor. 5.25 in $\mathcal{O}(\ell)$ time.

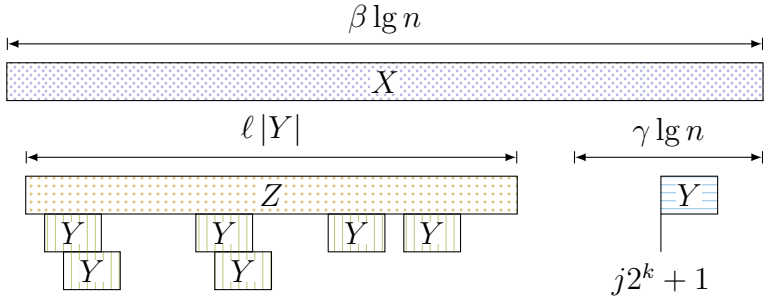


Fig. 5.36: Setting of Lemma 5.28. Given a string X of length $\beta \lg n$ endowed with the data structure described in Lemma 5.28, this data structure can find all occurrences of the rightmost k-basic factor Y starting at a position $j2^k + 1$ for an integer j. The difference to Lemma 5.26 is that Y has to appear in the last $\gamma \lg n$ characters of X.

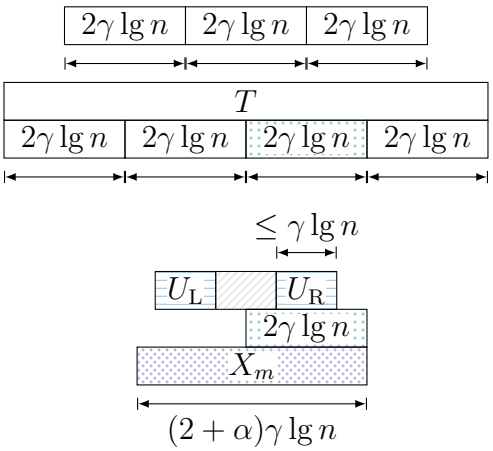


Fig. 5.37: Idea of the proof of Lemma 5.29. The algorithm iterates over m, and uses the data structures built on each superblock X_m to spot gapped repeats whose left and right arms are contained X_m while the right arms are at the last $\gamma \lg n$ positions of X_m.

Algorithm 17: Scaffold of the proof of Lemma 5.29 computing all maximal α-gapped repeats with an arm length between 1 and $\gamma \lg n$.

```
1 foreach 0 ≤ m ≤ n/lg n − γ − 1 do
2     compute data structure of Lemma 5.27 on superblock X_m
3     foreach 0 ≤ k ≤ lg(γ lg n) do
4         foreach basic
          factor Y_R = X_m[|X_m| − γ lg n + 1 + j2^k .. |X_m| − γ lg n + (j+1)2^k]
          with an integer j do
5             𝒴 ← query data structure of Lemma 5.27 on X_m with pattern
              Y_R
6             foreach Y_L ∈ 𝒴 do
7                 extend Y_L and Y_R to a gapped repeat with case
                  analysis (a)–(i)
```

5.4.3 Short Arms

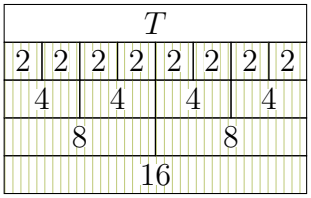
After having described all tools, we continue with the computation of all maximal α-gapped repeats, and focus on Set 3. The proofs of Lemmas 5.29, 5.31 and 5.32 adapt the techniques and the structure of the proof of [70, Thm. 7], where Dumitran et al. showed how to compute the longest α-gapped repeat in $\mathcal{O}(\alpha n)$ time. The main difference is that we take additional care in (a) reporting each maximal α-gapped repeat *only once*, and (b) speeding up the detection of *maximal* α-gapped repeats whenever parts of the arms are contained in a run to guarantee $\mathcal{O}(1)$ time per reported maximal α-gapped repeat.

Lemma 5.29. Given a string T and $\alpha \geq 1$, we can find all maximal α-gapped repeats $(U_{\mathrm{L}}, U_{\mathrm{R}})$ with $1 < |U_{\mathrm{R}}| \leq \gamma \lg n$ and $\mathsf{e}(U_{\mathrm{L}}) < \mathsf{b}(U_{\mathrm{R}})$ occurring in T, in $\mathcal{O}(\alpha n)$ time with $\mathcal{O}(\alpha n)$ words of working space.

Proof. A maximal α-gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$ with $|U_{\mathrm{R}}| \leq \gamma \lg n$ has a right arm U_{R} that must be contained in a segment $T[m \lg n + 1 \,..\, (m + \gamma + 1) \lg n]$, for an integer m with $0 \leq m \leq \frac{n}{\lg n} - \gamma - 1$. By fixing the interval where U_{R} can occur (i.e., fix m), we know that the entire repeat is contained in X_m (see Fig. 5.37).

An overview of our algorithm follows (see also Algo. 17): As a preprocessing step, we build $\mathsf{LCE}^{\leftrightarrow}$, and endow every superblock with the data structure described in Lemma 5.27. The last step takes $\mathcal{O}(\alpha\gamma n)$ time and $\mathcal{O}(\alpha\gamma n)$ words of space in total for all superblocks. For the actual search, we process each superblock linearly. In each superblock X_m, we search for each maximal α-gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$ whose arm U_{R} is contained in the suffix of length $\gamma \lg n$ of X_m (and hence $U_{\mathrm{L}} \subset X_m$). To spot the right arm U_{R} of a possible gapped repeat, we iterate over all possible lengths. Since a linear scan over all lengths would take too much time, we first compute an α-gapped repeat whose right arm is a basic factor, and then try to extend such an α-gapped repeat to a maximal α-gapped repeat.

In more detail, we iterate over the integer k with $0 \leq k \leq \lg(\gamma \lg n)$ to find α-gapped repeats with an arm length between 2^{k+1} and 2^{k+2}: For a fixed k, we partition the text into k-basic factors of length 2^k, and iterate over each such k-basic factor (there are $\gamma \frac{\lg n}{2^k}$ many). To avoid confusion, we call the elements of this partitioning *k-basic segments*.



Suppose that a k-basic segment Y_{R} is contained in the right arm of a maximal repeat, we retrieve its previous occurrences (i.e., all k-basic factors equal to this k-basic segment) such that each previous occurrence Y_{L} forms an α-gapped repeat $(Y_{\mathrm{L}}, Y_{\mathrm{R}})$. Subsequently, we extend both arms Y_{L} and Y_{R} to their left and to their right. The partitioning of the text into (non-overlapping) k-basic segments is sufficient to detect the right arms of all maximal gapped repeats, since we do not allow the arms to overlap (which belong to Set 2).

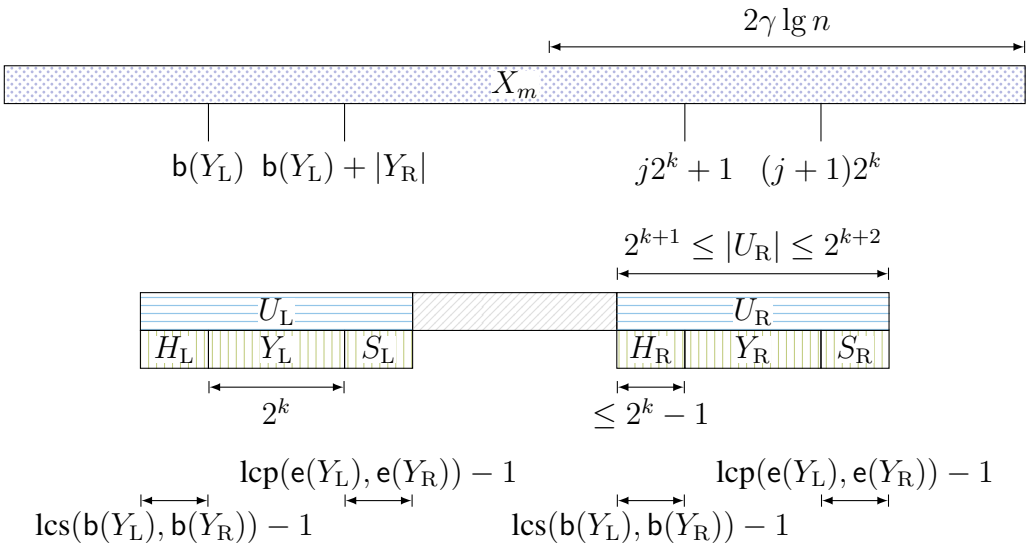
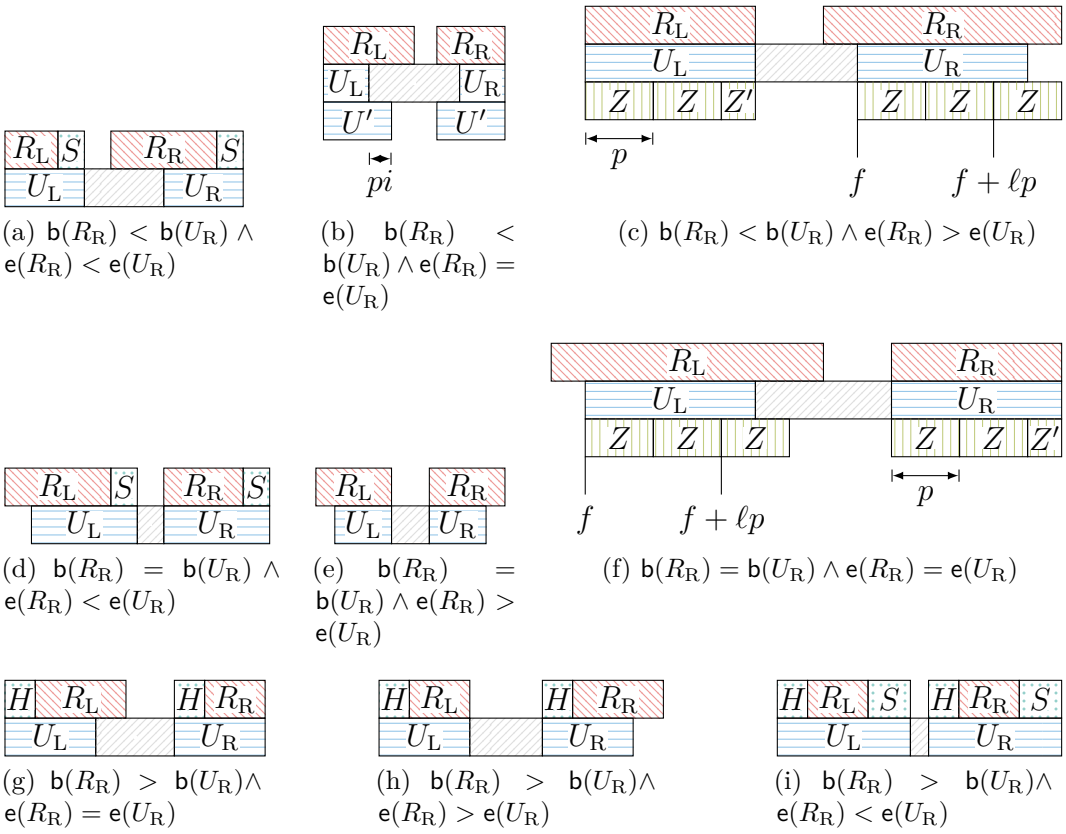


Fig. 5.38: Extending a gapped repeat whose right arm is a k-basic segment Y_R. Fixing X_m in the proof of Lemma 5.29, we try to spot gapped repeats whose arms contain an occurrence of Y_R. If we can extend this gapped repeat to a maximal gapped repeat, we output it. The prefix H_R has to be at most $2^k - 1$ characters long, since otherwise $(U_\mathrm{L}, U_\mathrm{R})$ will be found with the next k-basic segment $T[(j+1)2^k + 1 \,..\, (j+2)2^k]$.

We start with fixing a superblock X_m. We build the maximal α-gapped repeats whose right arms are contained in the last $2\gamma \lg n$ positions of X_m by extending gapped repeats whose arms are basic factors (see Fig. 5.38). A maximal α-gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ with $2^{k+1} \le |U_\mathrm{R}| \le 2^{k+2}$ has a right arm U_R that contains at least one segment $Y_\mathrm{R} = T[j2^k + 1 \,..\, (j+1)2^k]$ starting within the first 2^k positions of U_R ($\mathsf{b}(U_\mathrm{R}) \le \mathsf{b}(Y_\mathrm{R}) < \mathsf{b}(U_\mathrm{R}) + 2^k$). By definition, there is an occurrence Y_L of the segment Y_R that occurs also within the first 2^k positions of U_L, namely $Y_\mathrm{L} = T[\mathsf{b}(U_\mathrm{L}) + \mathsf{b}(Y_\mathrm{R}) - \mathsf{b}(U_\mathrm{R}) \,..\, \mathsf{b}(U_\mathrm{L}) + \mathsf{e}(Y_\mathrm{R}) - \mathsf{b}(U_\mathrm{R})]$. Finding the respective occurrence Y_L of Y_R helps us discovering the location of U_L.

Suppose that we identified the occurrence Y_L with $\mathsf{b}(Y_\mathrm{L}) < \mathsf{b}(Y_\mathrm{R})$. We try to build U_L and U_R by extending Y_L and Y_R to the left and to the right. To this end, we compute the LCS H ending at $j2^k$ at $\mathsf{b}(Y_\mathrm{L}) - 1$, and the LCP S starting at $(j+1)2^k + 1$ and at $\mathsf{e}(Y_\mathrm{L}) + 1$. We obtain $\mathsf{e}(U_\mathrm{L}) = \mathsf{b}(Y_\mathrm{L}) + |Y| + |S| - 1$ and $\mathsf{b}(U_\mathrm{R}) = j2^k + 1 - |H|$. If $\mathsf{e}(U_\mathrm{L}) \ge \mathsf{b}(U_\mathrm{R})$, then U_L and U_R overlap, and hence can be ignored (they are already found with Lemma 5.24). Otherwise, let S_L and S_R denote the left and right occurrences of S, and let H_L and H_R denote the left and right occurrences of H, respectively. Then U_L is obtained by concatenating H_L, Y_L, and S_L, while U_R is obtained by concatenating H_R, Y_R, and S_R. To avoid duplicates, the determined repeat is only reported if its right arm contains the position $\mathsf{b}(Y_\mathrm{R}) = j2^k + 1$ within its first 2^k positions.



(a) $\mathsf{b}(R_R) < \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) < \mathsf{e}(U_R)$

(b) $\mathsf{b}(R_R) < \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) = \mathsf{e}(U_R)$

(c) $\mathsf{b}(R_R) < \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) > \mathsf{e}(U_R)$

(d) $\mathsf{b}(R_R) = \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) < \mathsf{e}(U_R)$

(e) $\mathsf{b}(R_R) = \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) > \mathsf{e}(U_R)$

(f) $\mathsf{b}(R_R) = \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) = \mathsf{e}(U_R)$

(g) $\mathsf{b}(R_R) > \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) = \mathsf{e}(U_R)$

(h) $\mathsf{b}(R_R) > \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) > \mathsf{e}(U_R)$

(i) $\mathsf{b}(R_R) > \mathsf{b}(U_R) \wedge \mathsf{e}(R_R) < \mathsf{e}(U_R)$

Fig. 5.39: Spotting gapped repeats with periodicity in the case analysis of the proof of Lemma 5.29.

Otherwise we found (U_L, U_R) with the previous k-basic segment.

The algorithm above does not describe how to find the occurrence Y_L (efficiently). We rectify this omission now: Since $|U_R| < 2^{k+2}$ and $|Y_R| = 2^k$, the occurrence Y_L is contained in the segment $T[-\alpha 2^{k+2} + j2^k + 1 \,..\, j2^k] \subset X_m$ of length $\alpha 2^{k+2}$ ending at position $j2^k$. In our preprocessing, we already endowed X_m with the data structure of Lemma 5.27. We use this data structure as described in Lemma 5.28: It allows us to retrieve every possible segment Y_L inside the segment of length $\alpha 2^{k+2}$ ending at position $j2^k$, in $\mathcal{O}(\alpha)$ time. These occurrences are given in the representation of Cor. 5.25. They consist of single occurrences and the first occurrences within runs. There are $\mathcal{O}(\alpha)$ single occurrences, and we can process each single occurrence Y_L individually to find the maximal α-gapped repeat that is determined by Y_R and its respective occurrence Y_L.

However, it is not efficient to do the same for the occurrences of Y_L that are within a run (there can be $\Omega(\alpha)$ many occurrences). If Y_L is within a run R, Y_R is periodic. Let p be the smallest period of Y_R. Instead of examining all occurrences of Y_R in R, we focus on its first occurrence, since there are $\mathcal{O}(\alpha)$ many first

occurrences of Y_{R} in X_m. Let Y denote the substring of Y_{R}. Suppose that there is a repetition of Y's inside the segment $T[-\alpha 2^{k+2} + j2^k + 1 \,..\, j2^k] \subset X_m$. With Cor. 2.11 we can determine the run R_{L}. Similarly, we can determine the run R_{R} with smallest period p that contains Y_{R}.

Our goal is to spend $\mathcal{O}(\alpha + \mathrm{occ})$ time on all previous occurrences of Y_{R} in X, where occ is the number of all reported maximal α-gapped repeats while having X and Y_{R} fixed. We achieve this goal with a thorough case analysis, which is illustrated in Fig. 5.39. We group the cases depending on the relationship between $\mathsf{b}(U_{\mathrm{R}})$ and $\mathsf{b}(R_{\mathrm{R}})$. Each case is further differentiated into sub-cases according to the relationship between $\mathsf{e}(U_{\mathrm{R}})$ and $\mathsf{e}(R_{\mathrm{R}})$. In each sub-case, we determine the exact location of U_{L} and U_{R} by querying $\mathsf{LCE}^{\leftrightarrow}$ on X_m:
Cases (a-c): $\mathsf{b}(R_{\mathrm{R}}) < \mathsf{b}(U_{\mathrm{R}})$. Then $\mathsf{b}(U_{\mathrm{R}}) < \mathsf{e}(R_{\mathrm{R}})$, because we search arms with a length of at least two (arms of length one belong to Set 1). Then U_{L} starts at the first position of R_{L} (otherwise, we could extend both arms to the left, a contradiction to the maximality of the gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$).

(a) If U_{R} ends at a position to the right of R_{R}, then U_{L} ends at a position to the right of R_{L} (otherwise, it would contradict the maximality of R_{L} or R_{R}). The suffix $T[\mathsf{e}(R_{\mathrm{L}}) + 1 \,..\, \mathsf{e}(U_{\mathrm{L}})]$ of U_{L} occurring after the end of R_{L} is equal to the suffix $T[\mathsf{e}(R_{\mathrm{R}}) + 1 \,..\, \mathsf{e}(U_{\mathrm{R}})]$ of U_{R} occurring after the end of R_{R}. This common suffix is determined by the LCP starting at positions $\mathsf{e}(R_{\mathrm{L}}) + 1$ and $\mathsf{e}(R_{\mathrm{R}}) + 1$. Together with the length of R_{L} it determines $(U_{\mathrm{L}}, U_{\mathrm{R}})$.

(b) If U_{R} ends exactly at the same position as R_{R} ($\mathsf{e}(U_{\mathrm{R}}) = \mathsf{e}(R_{\mathrm{R}})$), then U_{R} is periodic with smallest period p (like R_{R}). We compute the longest prefix U' of R_{L} that is a suffix of R_{R}. By knowing the period p (determined by two subsequent occurrences of Y_{L}) and the length of R_{L} and R_{R}, the segment U' can be determined in constant time.[5]

Since U_{L} is longer than p, the α-gapped repeats under consideration have the left arm $U_{\mathrm{L}} := R_{\mathrm{L}}[1 \,..\, |U'| - pi]$ and the right arm $U_{\mathrm{R}} := R_{\mathrm{R}}[|R_{\mathrm{R}}| - (|U'| - pi) + 1 \,..\, |R_{\mathrm{R}}|]$ for $i \geq 0$ such that $(U_{\mathrm{L}}, U_{\mathrm{R}})$ is α-gapped. To obtain a time linear in the number of reported gapped repeats, we iterate over all values of i, starting at zero and stopping when the computed gapped repeat is no longer α-gapped.

(c) The final case is when U_{R} ends at a position of R_{R} prior to R_{R}'s last position ($\mathsf{e}(U_{\mathrm{R}}) < \mathsf{e}(R_{\mathrm{R}})$). In that case, we obtain that $U_{\mathrm{L}} = R_{\mathrm{L}}$ (otherwise, we could extend both arms to the right). The left arm U_{L} is equal to a substring $Z^h Z'$ for an integer $h \geq 2$, where $Z = R_{\mathrm{L}}[1 \,..\, p]$, p is the smallest period of R_{L}, and Z' is a prefix of Z.

We obtain the position of the first and the last occurrence of Z in R_{R} by an arithmetic progression with common difference p: If the first occurrence

[5] It can happen that $R_{\mathrm{L}} \equiv R_{\mathrm{R}}$, such that $U' \equiv R_{\mathrm{L}}$.

starts at f, then the starting positions of the succeeding occurrences of Z form the arithmetic progression $f, f+p, \ldots, f+\ell p$ for an integer $\ell \geq 1$. For each integer i with $0 \leq i \leq \ell$, we let U_{R} start at position $f+ip$ (and check whether $U_{\mathrm{R}} = U_{\mathrm{L}}$ by knowing the length of U_{R} and R_{R}).

Additional care has to be taken for the border case that U_{R} is a suffix of R_{R}. In this case, we have to check that we cannot extend simultaneously U_{R} and U_{L} to the right. The other border case that U_{R} is a prefix of R_{R} cannot happen, since we assumed in Cases (a-c) that $\mathsf{b}(U_{\mathrm{R}}) > \mathsf{b}(R_{\mathrm{R}})$.

Cases (d-f): $\mathsf{b}(U_{\mathrm{R}}) = \mathsf{b}(R_{\mathrm{R}})$.

(d) If U_{R} ends at a position to the right of R_{R}, then U_{L} ends after R_{L} ($\mathsf{e}(R_{\mathrm{R}}) < \mathsf{e}(U_{\mathrm{R}})$ and $\mathsf{e}(R_{\mathrm{L}}) < \mathsf{e}(U_{\mathrm{L}})$), and the suffix $T[\mathsf{e}(R_{\mathrm{L}})+1\,..\,\mathsf{e}(U_{\mathrm{L}})]$ of U_{L} occurring after R_{L} is equal to the suffix $T[\mathsf{e}(R_{\mathrm{R}})+1\,..\,\mathsf{e}(U_{\mathrm{R}})]$ of U_{R} occurring after R_{R}. This common suffix is equal to the LCP starting at the positions $\mathsf{e}(R_{\mathrm{L}})+1$ and $\mathsf{e}(R_{\mathrm{R}})+1$. Together with the length of R_{R} it determines $(U_{\mathrm{L}}, U_{\mathrm{R}})$.

(e) If U_{R} ends at a position inside R_{R} prior to its last position ($\mathsf{b}(R_{\mathrm{R}}) = \mathsf{e}(U_{\mathrm{R}}) < \mathsf{e}(R_{\mathrm{R}})$), then U_{L} ends at the last position of R_{L} (otherwise, both arms could be extended to the right). This means that the gap between the two arms is uniquely determined, and that the arms are periodic with smallest period p. We compute the longest suffix $T[\mathsf{e}(R_{\mathrm{L}})-\ell+1\,..\,\mathsf{e}(R_{\mathrm{L}})]$ of R_{L} that is a prefix of R_{R} (by knowing the starting positions of Y's occurrences in R_{L} and R_{R}), and check whether $(T[\mathsf{e}(R_{\mathrm{L}})-\ell+1\,..\,\mathsf{e}(R_{\mathrm{L}})]$, $T[\mathsf{b}(R_{\mathrm{R}})\,..\,\mathsf{b}(R_{\mathrm{R}})+\ell-1])$ is a maximal α-gapped repeat.

(f) If U_{R} ends at the last position of R_{R}, then we know the exact location of U_{R}. We can proceed analogously to Case (c) by symmetry.

Cases (g-i): $\mathsf{b}(U_{\mathrm{R}}) < \mathsf{b}(R_{\mathrm{R}})$. Since R_{L} and R_{R} are *maximal* repetitions, U_{L} starts at a position before the first position of R_{L} ($\mathsf{b}(U_{\mathrm{L}}) < \mathsf{b}(R_{\mathrm{L}}) < \mathsf{e}(U_{\mathrm{L}})$); the prefix of U_{R} occurring before the beginning of R_{R} is equal to the prefix of U_{L} occurring before R_{L}. The length of this common prefix can be retrieved with a longest common suffix query.

(g) Given $\mathsf{e}(U_{\mathrm{R}}) = \mathsf{e}(R_{\mathrm{R}})$, the length of the gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$ is determined by the common prefix and the length of R_{R}.

(h) The case $\mathsf{e}(U_{\mathrm{R}}) < \mathsf{e}(R_{\mathrm{R}})$ is symmetric to Case (g): Since $\mathsf{e}(U_{\mathrm{R}}) < \mathsf{e}(R_{\mathrm{R}})$, it holds that $\mathsf{e}(U_{\mathrm{L}}) = \mathsf{e}(R_{\mathrm{L}})$, and therefore the length of the gapped repeat is determined by their common prefix and the length of R_{L}.

(i) The last case is $\mathsf{e}(R_{\mathrm{R}}) < \mathsf{e}(U_{\mathrm{R}})$. Since R_{L} and R_{R} are maximal repetitions, we follow that $\mathsf{e}(R_{\mathrm{L}}) < \mathsf{e}(U_{\mathrm{L}})$. Consequently, U_{L} and U_{R} contain R_{L} and R_{R}, respectively. We determine the arms U_{L} and U_{R} by the LCP starting

at $\mathsf{e}(R_\mathrm{L}) + 1$ and $\mathsf{e}(R_\mathrm{R}) + 1$, and the longest common suffix ending at $\mathsf{b}(R_\mathrm{L}) - 1$ and $\mathsf{b}(R_\mathrm{R}) - 1$.

To sum up, we can determine the locations of both arms of each reported maximal α-gapped repeat in constant time for all the cases with LCE/LCS queries. For each found pair of arms, we have to check whether the arms form a valid maximal α-gapped repeat without overlap. To avoid duplicates, we additionally check whether

- the length of the arms is between 2^{k+1} and 2^{k+2} (otherwise we find these arms later with a k'-basic segment for $k' > k$), and whether
- the right arm contains position $j2^k + 1$ of X_m within its first 2^k positions (otherwise we find the right arm with the next k-basic segment $T[(j + 1)2^k + 1 \mathinner{..} (j + 2)2^k]$).

To ensure the last condition in Cases (b) and (c), where the right arm cannot be uniquely determined in general, we select those gapped repeats $(U_\mathrm{L}, U_\mathrm{R})$ with $\mathsf{b}(Y_\mathrm{R}) \in [\mathsf{b}(U_\mathrm{R}) \mathinner{..} \mathsf{b}(U_\mathrm{R}) + 2^k - 1]$.

This concludes our analysis for finding all α-gapped repeats of X_m, for each m *separately*. We can ensure that our algorithm finds and outputs each maximal repeat exactly once when processing X_{m+1} after having processed X_m: We check that the right arm of each found maximal α-gapped repeat is not completely contained in X_m (which means that we have already found it while processing X_m). To discard these gapped repeats, we modify our search as follows: When constructing the arms that are determined by a single occurrence of Y_R, we check the above containment condition separately; when constructing the arms determined by a run of Y_R-occurrences (cf. Cases (b) and (c)), we impose the condition that the right arm extends out of X_m when determining the starting positions of all possible arms. This is done by iterating over all possible starting positions of the right arm, beginning with the rightmost position, and stopping when the right arm is completely contained in X_m.

Finally, we analyze the complexity of the algorithm. We need $\mathcal{O}(n)$ preprocessing time for building $\mathsf{LCE}^{\leftrightarrow}$, and $\mathcal{O}(|X_m|) = \mathcal{O}(\gamma\alpha \lg n)$ preprocessing time for each X_m according to Lemma 5.27. For fixed m, k, and j (and hence fixing X_m and Y_R), our algorithm (finding all Y_L-candidates and performing the case analysis) takes $\mathcal{O}(\alpha + \mathrm{occ}_{m,k,j})$ time, where $\mathrm{occ}_{m,k,j}$ is the number of all maximal α-gapped repeats determined with the fixed values m, k, and j. Iterating over all values m, k and j gives the overall time complexity of the algorithm, which is order of

$$\underbrace{n}_{\text{precomp. on } T} + \underbrace{\sum_{m=0}^{\frac{n}{\lg n}}}_{\text{for all } X_m} \left(\underbrace{\gamma\alpha \lg n}_{\text{precomp. on } X_m} + \sum_{k=0}^{\lg(\gamma \lg n)} \left(\underbrace{\sum_{j=0}^{\gamma \frac{\lg n}{2^k}} (\alpha + \mathrm{occ}_{m,k,j})}_{\text{for all } k\text{-basic segments } Y_\mathrm{R}} \right) \right) =$$

$\mathcal{O}(\alpha n)$, since $\sum_{k=0}^{\lg(\gamma \lg n)} \sum_{j=0}^{\gamma \lg n/2^k} \alpha = \sum_{k=0}^{\lg(\gamma \lg n)} (1 + \gamma \lg n/2^k)\alpha = \mathcal{O}(\alpha \lg n)$, and the total number of maximal α-gapped repeats $\sum_{m=0}^{n/\lg n} \sum_{k=0}^{\lg(\gamma \lg n)} \sum_{j=0}^{\gamma \lg n/2^k} \mathrm{occ}_{m,k,j}$ is $\mathcal{O}(\alpha n)$ according to Thm. 5.16. □

5.4.4 Long Arms

With the superblocks X_m in Lemma 5.29 we can only find arms with a length of at most $\gamma \lg n$. Supporting longer arms would slow down the computation in Lemma 5.29 such that we need another trick to find the long arms. We borrow such a trick from Dumitran et al. [70, Thm. 7], where the authors introduced a block-representation of a string T: We partition T into segments of $\lg n$ characters $T[1 + i \lg n \mathinner{..} (i + 1) \lg n]$ for every $0 \leq i < n/\lg n$ (we can ensure that every block has the same number of characters by padding T with dummy characters such that $\frac{n}{\lg n}$ is integer). We call these segments *blocks* of T. The lexicographic order on Σ induces a linear order on the blocks. Since there are at most $n/\lg n$ different blocks, we can enumerate the blocks with ranks from 1 to at most $n/\lg n$ such that the j-th smallest block receives rank j. For our purpose, an enumeration from 1 to n (we omit some ranks) is sufficient. Before showing how to compute the enumeration, we start with the definition of the block-representation: A string $\widetilde{T}$ is the *block-representation* of T if

- $\widetilde{T}$ is a string of length $n/\lg n$ on the alphabet $\{1, \ldots, n\}$, and
- $\widetilde{T}[i] = j$ $(1 \leq i \leq n/\lg n)$ if and only if the block of T with rank j is equal to $T[1+(i-1)\lg n \mathinner{..} i \lg n]$; in this case we say that $T[1+(i-1)\lg n \mathinner{..} i \lg n]$ *corresponds* to $\widetilde{T}[i]$, and vice versa.

It is easy to provide a linear-time algorithm computing the enumeration: We start with building $\mathsf{LCE}^{\leftrightarrow}$. Subsequently, we cluster together the suffixes of the suffix array that share a common prefix of length at least $\lg n$. There are at most n different clusters; hence we can enumerate all clusters, starting from 1 to at most n. A block is associated with the number of a cluster if the cluster contains the suffix that starts at the same position as the block.

Lemma 5.30. We can build the block-representation $\widetilde{T}$ of a string T of length n in $\mathcal{O}(n)$ time.

Having Lemma 5.30, we are ready to present the algorithm finding long-armed maximal α-gapped repeats with large values for α:

Lemma 5.31. Given a string T of length n, and an $\alpha \geq \lg n$, we can find all maximal α-gapped repeats $(U_\mathrm{L}, U_\mathrm{R})$ with $|U_\mathrm{R}| > \gamma \lg n$ and $\mathsf{e}(U_\mathrm{L}) < \mathsf{b}(U_\mathrm{R})$ occurring in T, in $\mathcal{O}(\alpha n)$ time using $\mathcal{O}(\alpha n)$ words of working space.

Proof. We use similar techniques as presented in Lemma 5.29 to prove this lemma: We partition the text into segments, and iterate over each segment Y_R

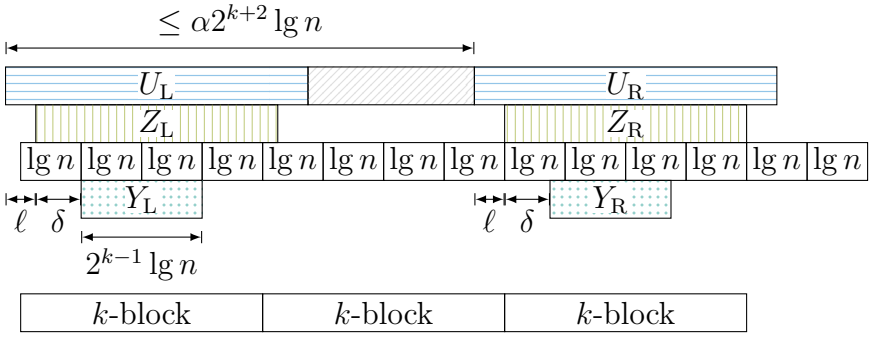


Fig. 5.40: Finding occurrences of Y starting with a block in proof of Lemma 5.31. The distance δ is smaller than $\lg n$, and the distance ℓ is smaller than $2^k \lg n$, since Z is the first k-block contained in Z.

of this partition, trying to build a gapped repeat with Y_{R} as its right arm. The main difference is that we cannot use the result of Lemma 5.27, since we deal with gapped repeats with arms longer than $\gamma \lg n$. Instead, we use the data structures described in Lemma 5.26. However, to obtain the stated complexity, we apply Lemma 5.26 to the block-representation of T, rather than to T itself.

In this sense, the first step is to construct the block-representation $\widetilde{T}$ of T. Subsequently, we construct $\mathsf{LCE}_T^{\leftrightarrow}$ and $\mathsf{LCE}_{\widetilde{T}}^{\leftrightarrow}$, as well as the data structure of Lemma 5.26 for the string $\widetilde{T}$. Finally, we compute all runs of T with Lemma 2.2. Every step is conducted in $\mathcal{O}(n)$ time.

Like in the proof of Lemma 5.29, we iterate over all possible arm lengths. For an integer k with $\lg \gamma - 1 \leq k \leq \lg(n/\lg n) - 2$, we search for all maximal α-gapped repeats $(U_{\mathrm{L}}, U_{\mathrm{R}})$ in T with $2^{k+1} \lg n \leq |U_{\mathrm{L}}| \leq 2^{k+2} \lg n$.

For the following, we fix k. We partition the string T into segments. Each segment has the length $2^k \lg n$, and is called a *k-block*. As for blocks, we assume that each k-block has the same number of characters (by padding the input text). The idea of this partition is the following: If a maximal α-gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$ with $2^{k+1} \lg n \leq |U_{\mathrm{L}}| \leq 2^{k+2} \lg n$ exists, then it contains a k-block within its first $2^k \lg n$ positions. Let Z_{R} be the *first* k-block contained in U_{R}. Since U_{R} contains Z_{R}, the left arm U_{L} also contains an occurrence Z_{L} of Z_{R}. However, Z_{L} is *not necessarily* starting at a position $j \lg n + 1$ for an integer $j \geq 0$, i.e., it does not have to start with a block. In general, we cannot capture Z_{L} with our block-representation. Fortunately, at least one of the segments Y_{R} of length $2^{k-1} \lg n$ starting within the first $\lg n$ positions of Z_{R} has an occurrence Y_{L} in U_{L} starting with a block (Y_{R} itself does *not* have to start with a block, see also Fig. 5.40). To find such a segment Y_{R}, we iterate over the first $\lg n$ positions of Z_{R}. Let us fix a segment Y_{R} of length $2^{k-1} \lg n$ that starts within the first $\lg n$ positions of Z_{R}. Since we search for an α-gapped repeat $(U_{\mathrm{L}}, U_{\mathrm{R}})$ with (a) $\mathsf{b}(U_{\mathrm{R}}) - \mathsf{b}(U_{\mathrm{L}}) \leq \alpha |U_{\mathrm{R}}| \leq \alpha 2^{k+2} \lg n$, (b) $Y_{\mathrm{L}} \subset U_{\mathrm{L}}$, (c) $Y_{\mathrm{R}} \subset U_{\mathrm{R}}$, and (d) Z_{R} starts within the first $2^k \lg n$ positions of U_{R}, we look for the occurrences Y_{L}

of Y_R starting at one of the $2^k \lg n + \alpha 2^{k+2} \lg n = (4\alpha + 1)2^k \lg n$ positions to the left of Z_R. For each such occurrence Y_L that corresponds to a sequence of blocks, we try to extend both Y_L and Y_R to an α-gapped repeat (which is the same strategy as used in the proof of Lemma 5.29).

Since Y_R is not necessarily a sequence of 2^{k-1} blocks, i.e., Y_R is not represented by a segment of $\widetilde{T}$ in general, we search for a sequence of blocks $\widetilde{Y}$ in $\widetilde{T}$ that corresponds to Y_R with the suffix array of $\widetilde{T}$. With this search we can figure out the characters contained in $\widetilde{Y}$ (speaking of the characters induced by the enumeration of the block-representation) if there is an occurrence of $\widetilde{Y}$ in $\widetilde{T}$. Let Y denote the substring of Y_R. By binary searching the suffix array of $\widetilde{T}$ (using LCE queries on T to compare the factors of $\lg n$ characters of Y and the blocks of $\widetilde{T}$, at each step of the search) we try to detect a substring of $\widetilde{T}$ that encodes a string equal to Y. Suppose that we can find such a sequence $\widetilde{Y}$ of blocks in $\widetilde{T}$ (otherwise, Y cannot correspond to a sequence of blocks from U_L, so we choose a new Y_R by taking the next starting position). By Lemma 5.26, we can spot the occurrences of $\widetilde{Y}$ in the $(4\alpha+1)2^k$ blocks of $\widetilde{T}$ that occur before the blocks of Z_R, in $\mathcal{O}(\lg\lg\left|\widetilde{T}\right| + \alpha)$ time; this range corresponds to $T[\mathsf{e}(Z_R) - (4\alpha+1)2^k \mathinner{..} \mathsf{e}(U_R) - 1]$. Each of those occurrences of $\widetilde{Y}$ fixes a possible left arm U_L. We can construct this arm U_L together with its corresponding arm U_R with the same techniques as in Lemma 5.29:

- In the case of a single occurrence $\widetilde{Y_L}$ (there are at most $\mathcal{O}(\alpha)$ many of such single occurrences), $\widetilde{Y_L}$ represents a single occurrence Y_L of Y. We extend Y_L and Y_R in both directions to obtain two arms U_L and U_R, respectively, for which we have to check if they define a valid α-gapped repeat (U_L, U_R) without overlaps. To avoid duplicates, we check that the length of each arm is between 2^{k+1} and 2^{k+2}, and that Z_R is the first k-block of the right arm.

- If an occurrence $\widetilde{Y_L}$ of $\widetilde{Y}$ is within a run, then $\widetilde{Y}$ is periodic. Given that p is $\widetilde{Y}$'s period, the smallest period of Y is at most $p \lg n \leq |Y|/2$, since $p \leq \widetilde{Y}/2 = Y/(2 \lg n)$. This means that Y is periodic, too. We can determine the actual period of Y with the computed runs of T analogously to Lemma 5.24, since we linearly scan the text from left to right such that we know the run containing Y_R. Knowing the smallest period of Y_R, we can compute the run R_L containing the segment Y_L that has $\widetilde{Y_L}$ as its block-representation (see Cor. 2.11). Finally, we conduct the case analysis as described in the proof of Lemma 5.29 on Y_R and the run R_L.

It remains to prove that each maximal gapped repeat is counted only once:

- If Y_R is within a run R_R with smallest period p, we skip the starting positions $T[\mathsf{b}(Y_R)+p \mathinner{..} \mathsf{e}(R_R)-|Y_R|+1]$ (segments starting at these positions with a length of $|Y_R| = 2^{k-1} \lg n$ have the same block-representation as

the segments of the same length starting p positions earlier). Every other two segments Y_R and $\overline{Y_\mathrm{R}}$ starting within the first $\lg n$ characters of Z_R cannot have the same block-representation, and therefore, they cannot determine the same Y_L-occurrences. Nevertheless, if they are periodic, they can determine the *same run* of Y_L-occurrences. To handle such a case, we mark the starting positions of all already visited runs in a bit vector (which gets cleared after processing Z_R), such that we can skip a run of Y_L-occurrences whenever it is already marked.

- On finding a segment Y_R occurring in the first $\lg n$ characters of a k-block Z_R such that Y_R determines a maximal α-gapped repeat, the same maximal α-gapped repeat will not be found by a segment with the same length as Y_R contained in another k-block (than Z_R), since Z_R is the *first* k-block of U_R.

Finally, we analyze the complexity of the above described algorithm. The preprocessing, i.e., the construction of $\widetilde{T}$ and all of the needed data structures, takes $\mathcal{O}(n)$ time. We have multiple nested iterations:

(a) We iterate over all integers k with $\lg\gamma - 1 \leq k \leq \lg(n/\lg n) - 2$ to find all maximal α-gapped repeats with arms having a length between $2^{k+1}\lg n$ and $2^{k+2}\lg n$.

(b) For a fixed integer k, we examine every k-block Z_R, and there are $n/(2^k \lg n)$ many.

(c) For a fixed segment Z_R, we analyze each segment Y_R of length $2^{k-1}\lg n$ starting within the first $\lg n$ positions of the chosen k-block Z_R.

(d) For each such segment Y_R we compute its block-representation $\widetilde{Y}$ with a search in the suffix array taking $\mathcal{O}(\lg(n/\lg n))$ time.

(e) Given that $\widetilde{Y}$ occurs in $\widetilde{T}$, we find all occurrences of $\widetilde{Y}$ that are possible substrings of the left arm of an α-gapped repeat in $\mathcal{O}(\lg\lg n + \alpha)$ time.

(f) For each Y_L of the $\mathcal{O}(\alpha)$ single occurrences, we check whether it is possible to extend Y_L and Y_R to a maximal α-gapped repeat in $\mathcal{O}(1)$ time. We also have $\mathcal{O}(\alpha)$ occurrences of the block encoding Y_L within runs, all of them are processed in $\mathcal{O}(\alpha + \mathrm{occ}_{Z_\mathrm{R},Y_\mathrm{R}})$ time overall, where $\mathrm{occ}_{Z_\mathrm{R},Y_\mathrm{R}}$ is the number of maximal α-gapped repeats we find for given segments Z_R and Y_R.

Overall, this adds up to

$$\sum_{k=\lg\gamma-1}^{\lg\frac{n}{\lg n}-2} \underbrace{\frac{n}{2^k\lg n}}_{\#Z_\mathrm{R}} \underbrace{\lg n}_{\#Y_\mathrm{R}} \underbrace{\left(\lg\frac{n}{\lg n} + \lg\lg n + \alpha + \mathrm{occ}_{Z_\mathrm{R},Y_\mathrm{R}}\right)}_{\text{process every possible } Y_\mathrm{L}} = \mathcal{O}(n\lg n + \alpha n), \tag{5.17}$$

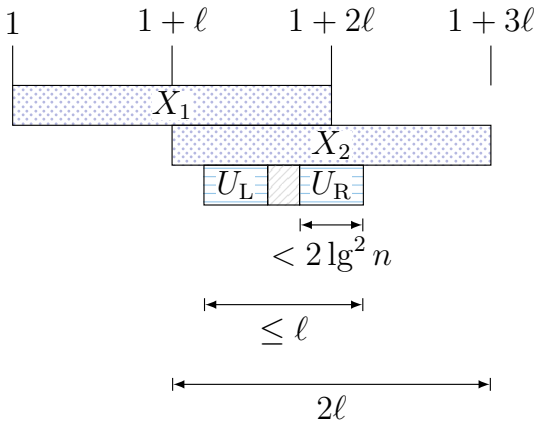


Fig. 5.41: Covering $\{X_m\}_m$ of T defined as in the proof of Lemma 5.32. Each X_m has a length of 2ℓ and starts at the text position $1+m\ell$. A gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$ with $\mathsf{e}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L}) + 1 \leq \ell$ is completely contained in one segment X_m of the cover.

since the total number of maximal α-gapped repeats in T is upper bounded by $\mathcal{O}(\alpha n)$. Since $\alpha \geq \lg n$, the statement of the lemma follows. □

Unfortunately, the algorithm presented in Lemma 5.31 does not run in $\mathcal{O}(\alpha n)$ time for $\alpha = o(\lg n)$, but rather in $\mathcal{O}(n \lg n)$ time. The following lemma shows how to achieve the $\mathcal{O}(\alpha n)$ time bound for $\alpha < \lg n$, with a trick presented (again) in [70, Thm. 7]. Remembering that we partition the text T into k-blocks in Lemma 5.31 to find all maximal α-gapped repeats with an arm length between $2^{k+1} \lg n$ and $2^{k+2} \lg n$, the idea is to apply Lemma 5.31 for large values of k, then apply Lemma 5.31 on a cover of the text, and finally Lemma 5.29 for small values of k.

Lemma 5.32. Given a string T of length n, and an $\alpha < \lg n$, we can find all maximal α-gapped repeats $(U_\mathrm{L}, U_\mathrm{R})$ with $|U_\mathrm{R}| > \gamma \lg n$ and $\mathsf{e}(U_\mathrm{L}) < \mathsf{b}(U_\mathrm{R})$ occurring in T, in $\mathcal{O}(\alpha n)$ time.

Proof. Initially, we run the algorithm of Lemma 5.31 only for $k > \lg \lg n$ to find all maximal α-gapped repeats with an arm length of at least $2^{\lg \lg n + 1} \lg n$. We see that Eq. (5.17) with $k > \lg \lg n$ yields $\mathcal{O}(\alpha n)$ time.

In the rest of this proof, we search for maximal α-gapped repeats whose arms' length is upper bounded by $2^{\lg \lg n + 1} \lg n = 2 \lg^2 n$. Setting $\ell := \alpha \cdot 2 \lg^2 n + 2 \lg^2 n = 2(\alpha + 1) \lg^2 n$, the lengths of those gapped repeats (i.e., $\mathsf{e}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L}) + 1$ for a gapped repeat $(U_\mathrm{L}, U_\mathrm{R})$) is at most ℓ. If we cover T with the set of segments $\{T[1 + m\ell \mathinner{..} (m+2)\ell] \mid 0 \leq m \leq n/\ell - 2\}$, then such an α-gapped repeat is contained in (at least) one segment of this cover (see Fig. 5.41).

Having this cover, we can apply the algorithm of Lemma 5.31 to each segment in the cover (iterating over all m) to detect all maximal α-gapped repeats with an arm length of *at least* $2^{\lg \lg (2\ell) + 1} \lg(2\ell)$ and less than $2^{\lg \lg n + 1} \lg n$ that are completely contained in a segment of the cover. Given that occ_m is the number of occurrences of all those maximal α-gapped repeats in the m-th segment of the cover, Equation (5.17) with $k \geq \lg \lg (2\ell)$ gives $\mathcal{O}(\alpha\ell + \mathrm{occ}_m)$ time for the

algorithm of Lemma 5.31 running on the m-th segment of the cover, since each cover is of length 2ℓ, and we build the suffix array and the block representation on each cover, such that n becomes 2ℓ in Eq. (5.17). Summing over all segments of the cover, we get $\mathcal{O}(\alpha n)$ time in total. By knowing the overlap of two subsequent segments of the cover, it is easy to adapt the algorithm of Lemma 5.31 in such a way that no gapped repeat is reported twice.

It is left to find all maximal α-gapped repeats with an arm length smaller than $2^{\lg\lg(2\ell)+1}\lg(2\ell)$. For n large enough, it holds that $2^{\lg\lg(2\ell)+1}\lg(2\ell) \leq \gamma \lg n$, since $\alpha \leq \lg n$. But those maximal α-gapped repeats are already found by the algorithm of Lemma 5.29 running in $\mathcal{O}(\alpha n)$ time. □

Putting the results of Lemmas 5.23, 5.24, 5.29, 5.31 and 5.32 together yields the following theorem:

Theorem 5.33. Given a string T and an $\alpha \geq 1$, we can compute $\mathcal{R}_\alpha(T)$ in $\mathcal{O}(\alpha n)$ time with $\mathcal{O}(\alpha n)$ words of working space.

Computing $\mathcal{P}_\alpha(T)$ can be done very similarly, as can be seen by the highly similar proofs of [70, Thm. 7] and [70, Thm 8].

Corollary 5.34. Given a string T and $\alpha > 1$, we can compute $\mathcal{P}_\alpha(T)$ in $\mathcal{O}(\alpha n)$ time.

Proof. We construct the data structure of Cor. 2.13 to test in constant time whether a substring $T[i\,..\,j]^\intercal$ occurs at a position in T. We split T into blocks and k-blocks (like in Lemma 5.31) for each $k \leq \lg|T|$, to check whether there exists a gapped palindrome $(U_\mathrm{L}, U_\mathrm{R})$ with $2^k \leq |U_\mathrm{R}| \leq 2^{k+1}$. This search is conducted analogously to the case of gapped repeats, with the difference that when fixing the segment Y_R in U_R, we look for the occurrences Y_L with $Y_\mathrm{L} = Y_\mathrm{R}{}^\intercal$ (instead of $U_\mathrm{L} = Y_\mathrm{R}$) in the segment of length $\mathcal{O}(\alpha 2^{k+1})$ preceding it. To find gapped palindromes with long arms, we compute the block-representation of $Y_\mathrm{R}{}^\intercal$ with the suffix array of $TT^\intercal$. □

The case $\alpha = 1$ is already solved in Lemma 2.4. Corollary 5.34 generalizes the result of Kolpakov and Kucherov [163], who presented an algorithm working on an alphabet with constant size.

Remark 5.35. The presented bounds are still valid when working with the more general definition of α-gapped φ-repeats or α-gapped φ-palindromes: Let $\varphi : \Sigma^* \to \Sigma^*$ be a string isomorphism, i.e., $\varphi(SV) = \varphi(S)\varphi(V)$, and φ is bijective. A pair of segments $(U_\mathrm{L}, U_\mathrm{R})$ with $\mathsf{e}(U_\mathrm{L}) + 1 \leq \mathsf{b}(U_\mathrm{R})$ is called a gapped φ-repeat (resp. gapped φ-palindrome) if $U_\mathrm{L} = \varphi(Y_\mathrm{R})$ (resp. $U_\mathrm{L} = \varphi(Y_\mathrm{R}{}^\intercal) = \varphi(Y_\mathrm{R})^\intercal$). The properties α-gapped and maximal are defined analogously to the gapped repeats (resp. gapped palindromes). Since the identity is a string isomorphism, gapped φ-repeats and gapped φ-palindromes are a generalization of gapped repeats and gapped palindromes without overlaps, respectively. It is easy to

see that our results are also applicable for α-gapped φ-repeats or α-gapped φ-palindromes. This generalizes the analysis in [163, Sec. 5]; there, φ is equal to a function building the base complements of a DNA string (the Watson-Crick base complement as in Fig. 5.1). The problem of enumerating all 1-gapped φ-repeats or all 1-gapped φ-palindromes was already investigated in [105, 106].

5.5 Conclusion

We presented two major achievements that shed light on the combinatorial and computational aspects of α-gapped repeats. First, we succeeded in giving the concrete upper bounds $3(\pi^2/6+5/2)\alpha n$ and $7(\pi^2/6+1/2)\alpha n - 3n - 1$ on the maximum number of all maximal α-gapped repeats $\mathcal{R}_\alpha(T)$ and on all maximal α-gapped palindromes $\mathcal{P}_\alpha(T)$, respectively, of a text T of length n. Second, we elaborated an algorithm computing the set $\mathcal{R}_\alpha(T)$ of a text T of length n whose characters are drawn from an integer alphabet. The algorithm runs in $\mathcal{O}(\alpha n)$ time, and can be adapted to compute the set $\mathcal{P}_\alpha(T)$. The combinatorial bounds and the time bounds of the algorithms are asymptotically optimal. Our proofs work for both supporting overlaps and prohibiting overlaps, and thus generalize the analysis of former studies. Our study does not lead to a dead end, as can be seen by the following open problems:

Space efficient computation. Throughout Sect. 5.4, we deliberately omitted the exact memory consumption of the created data structures (currently $\mathcal{O}(\alpha n)$ words). Current approaches for succinct LCE query data structures (replacing Cor. 2.12 with a data structure of Fig. 4.2) shed a positive light on lowering the space requirements. With a more careful analysis of the space we could give refined bounds (e.g., measured in bits) of the selected data structures, perhaps devising an algorithm working on succinct space.

Output-sensitive running time. It is also interesting to further refine both algorithms to such an extent that their running time is output-sensitive, i.e., having $\mathcal{O}(|T|+|\mathcal{R}_\alpha(T)|)$ and $\mathcal{O}(|T|+|\mathcal{P}_\alpha(T)|)$ worst case running time, respectively, for a string T.

Online algorithm. To the best of our knowledge, there is no efficient algorithm for computing all maximal α-gapped repeats/palindromes of a given string *online*. We are aware of the algorithm of Fujishige et al. [99] finding all gapped palindromes with a fixed gap ($\mathsf{b}(U_\mathrm{R}) - \mathsf{e}(U_\mathrm{L}) - 1 = g$ for a constant g) in $\mathcal{O}(n \lg \sigma)$ time online while taking $\mathcal{O}(n)$ words of working space.

Compression based on α-gapped repeats and palindromes. Already suggested by Chairungsee [43] or Crochemore et al. [64, Thm. 4], it would be

interesting to implement a compressor that considers gapped palindromes for compressing redundancy. In the setting of α-gapped repeats or palindromes, we could devise a compressor based on the LZ77 factorization. Like the coding defined in Sect. 3.3.4, we can replace a substring F with a referred position j and a length ℓ such that $F = T[j \,..\, j+\ell-1]$ (resp. $F = T[j \,..\, j+\ell-1]^{\mathsf{T}}$). The referred position j and the length ℓ are stored in a tuple (j, ℓ) in the compressed file such that a decompressor can restore the original text based on the values stored in each tuple. Usually, LZ77 compressors store the referred position j relatively to the starting position $\mathsf{b}(F)$, i.e., $\mathsf{b}(F) - j$ instead of j, if the referred positions are coded with a coder favoring small numbers (cf. the rightmost parsing problem in Sect. 3.9). The value $\mathsf{b}(F) - j$ can be upper bounded by $\alpha\,|F|$ if we restrict $(T[j \,..\, j+\ell-1], F)$ to be an α-gapped repeat (resp. palindrome). Informally, the inequality $\mathsf{b}(F) - j \leq \alpha\,|F|$ is a trade-off between the gain of the factor (the $|F|$ characters of F can be replaced by a reference) and the cost for storing the referred position. This trade-off is influenced by the choice of the parameter α. Additionally, instead of choosing the longest α-gapped repeat or palindrome, it would be interesting to apply ideas similar to [76] for aiming at multiple goals (like compression ratio, (de-)/compression speed, etc.).

Distinct sets. From the literature it is already known that searching all distinct squares (see Sect. 3.5) or all distinct ordinary palindromes [119] of a string of length n can be done in $\mathcal{O}(n)$ time. An extension is computing all distinct α-gapped repeats/palindromes, for which we are unaware of any results, both on the combinatorial (like giving an upper bound on the number of all distinct α-gapped repeats/palindromes) and on the algorithmic aspects. We call a set of gapped repeats (resp. palindromes) *distinct*, if the set does not contain two gapped repeats (resp. palindromes) $(U_{\mathrm{L}}, U_{\mathrm{R}})$ and $(\overline{U_{\mathrm{L}}}, \overline{U_{\mathrm{R}}})$ with $U_{\mathrm{L}} = \overline{U_{\mathrm{L}}}$.

Generalizing gaps. A generalization of α-gapped repeats are (μ, ν)-gapped repeats, i.e., gapped repeats $(U_{\mathrm{L}}, U_{\mathrm{R}})$ with the additional property that $\nu(|U_{\mathrm{L}}|) \leq \mathsf{b}(U_{\mathrm{R}}) - \mathsf{e}(U_{\mathrm{L}}) - 1 \leq \mu(|U_{\mathrm{L}}|)$ for two functions $\mu, \nu : \mathbb{N} \to \mathbb{R}$. The (μ, ν)-gapped repeats with $\mu(j) \coloneqq 1, \nu(j) \coloneqq \alpha j$ are exactly the α-gapped repeats without overlap. Kolpakov [159] showed that the number of all maximal (μ, ν)-gapped repeats is bounded by

$$\mathcal{O}(n(1 + \max(\sup_{j\in\mathbb{N}}(1/j)(\mu(j) - \nu(j)), \sup_{j\in\mathbb{N}} |\mu(j+1) - \mu(j)|\,, \sup_{j\in\mathbb{N}} |\nu(j+1) - \nu(j)|))).$$

Improving the upper bound, or devising a lower bound for certain μ and ν is left for future work.

Regarding the algorithmic part, Brodal et al. [37] presented an algorithm computing all maximal (μ, ν)-gapped repeats in $\mathcal{O}(n \lg n + \mathrm{occ})$ time, where occ is the number of occurrences. In the light that we achieved $\mathcal{O}(\alpha n)$ running

time for finding all maximal α-gapped repeats, it looks feasible to devise an algorithm whose running time depends linearly on n and on the values of μ and ν. Needless to say, (μ, ν)-gapped palindromes are also an unexplored topic.

Chapter 6

Epilogue

In the current information-oriented society, efficiently processing massive volumes of data that grow daily is a key topic. Specifically, we need to develop technologies that can compress large volumes of data to make data burdens as light and manageable as possible, while also enabling high-speed search. Achieving both goals isn't easy... Technology that decompresses only the portions of the data actually needed can help resolve these issues.

— Kunihiko Sadakane [218]

Sophisticated ways of handling regular structures lead to efficient algorithms and data structures working on texts. For instance, by handling discovered gapped repeats sophisticatedly, we could bound the running time of sorting suffixes with a naïve string sorter in Chapter 4. Similar concepts can be applied to devising index data structures based on the LZ77 (e.g., [102, 142]) or LZ78 (e.g., [13, 219]) factorization.

In this thesis, we worked with succinct and compact data structures. Their field of application is limited, since their space consumption is at least linear in the number of characters of their input text. However, *compressed* representations of the text or *compressed* data structures do not necessarily scale with the number of the characters. Instead, they can take a small fraction of space of the original input. The spotlight of current research moves towards highly repetitive texts (e.g., [103, Sect. 1]). It is reasonable to analyze algorithms and data structures to work in *compressed* space, or to work on *compressed* texts. Popular models are counting space or time consumption in the number of LZ77 factors, in the size of a grammar, or the size of the run-length encoding of the text or its BWT (see again [103, Sect. 1] for a survey on the connections between those numbers). It is left open whether we can take ideas from our presented algorithms to devise algorithms working within compressed space or working on compressed texts, for which several solutions are already known for finding regular structures (e.g., [131, 133]).

Symbol Register

Abbreviation	Written-Out
Algo.	Algorithm
Cor.	Corollary
Def.	Definition
Eq.	Equation
Fig.	Figure
Sect.	Section
Thm.	Theorem
cf.	compare to
e.g.	for example
i.e.	that is

Symbol	Meaning
$\perp$	invalid entry
ϵ	real constant, $0 < \epsilon \leq 1$
Λ	the empty string
$\mathcal{I}, \mathcal{J}, \mathcal{K}$	intervals
n	text size ($= \lvert T \rvert$)
p	period
σ	alphabet size
Σ	(usually an integer) alphabet
τ	trade-off constant
T	input text

Left: Register of reference abbreviations (*top*) and abbreviated words with Latin origin (*bottom*). *Right*: Register of common variables.

Function	Meaning
$\oplus$	bitwise exclusive OR
&	bitwise AND
$B.\mathsf{rank}_1(j)$	$\sum_{i=1}^{j} B[i]$
$B.\mathsf{select}_1(k)$	$\min\left\{j \in \mathbb{N} \mid \sum_{i=1}^{j} B[i] = k\right\}$
$\exp(S)$	exponent of a string S
$\mathcal{E}(T)$	$\sum_{R \text{ run of } T} \exp(R)$
$H_k(T)$	k-th order empirical entropy of $T \in \Sigma^*$ for an integer $k \geq 0$
$\lg n$	binary logarithm of n
$\lg^* n$	iterated logarithm $\min\left\{k \in \mathbb{N} \mid \lg^{(k)}(n) \leq 1\right\}$

Register of functions.

Symbol	Meaning
$\mathbb{N}$	natural numbers starting with one
$\mathbb{Z}$	integer numbers
$\mathbb{R}$	real numbers

Register of number sets.

Data Structure	Definition
BWT	BWT of T: $\mathsf{BWT}[i] = T[(\mathsf{SA}[i] + n - 2 \mod n) + 1]$
ISA	inverse suffix array of T: $\mathsf{SA}[\mathsf{ISA}[i]] = i$
$\mathsf{LCE}_T^{\leftrightarrow}$	data structure answering LCE queries on T and $T^\intercal$
LCP	LCP array of T with $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = \mathsf{lcp}(T[\mathsf{SA}[i] \mathinner{..} n], T[\mathsf{SA}[i-1] \mathinner{..} n])]$ for all $i = 2, \ldots, n$
Ψ	Ψ-function of T: $\Psi[i] = \mathsf{ISA}[(\mathsf{SA}[i] \mod n) + 1]$
PLCP	permuted LCP array of T: $\mathsf{PLCP}[\mathsf{SA}[i]] = \mathsf{LCP}[i]$
SA	suffix array of T: $T[\mathsf{SA}[i-1] \mathinner{..} n] \prec T[\mathsf{SA}[i] \mathinner{..} n]$ $\forall i = 2, \ldots, n$

Register of text data structures.

Notation	Meaning
$T[i \mathinner{..} j]$	substring from position i to j (inclusively) of a string T
$[i \mathinner{..} j]$	integer interval $\{i, \ldots, j\}$ for integers i, j with $i \leq j$.
$\mathsf{b}([i \mathinner{..} j])$	beginning position i
$\mathsf{e}([i \mathinner{..} j])$	ending position j
$\lvert T \rvert$	either the cardinality of a set T, or the length of the string T
$\lvert \mathcal{I} \rvert$	$\mathsf{e}(\mathcal{I}) - \mathsf{b}(\mathcal{I}) + 1$
$(f \circ g)(x)$	$f(g(x))$
$f^{(k)}(x)$	k-th iterated application of f to x: $\underbrace{(f \circ \cdots \circ f)}_{k \text{ times}}(x)$
$f^k(x)$	$(f(x))^k$, especially $\lg^k x = (\lg x)^k$
Σ^k	set of strings of length k drawn from the alphabet Σ
$a \leq i_1, \ldots, i_j \leq b$	$a \leq i_k \leq b$ $\forall k = 1, \ldots, j$
$T \equiv S$	T and S are the same segments
$S \prec T$	$S \in \Sigma^*$ is lexicographically smaller than T
$x \coloneqq y$ or $y =: x$	assign x the value y
$T^\intercal$	reverse of T: $T[\lvert T \rvert] \cdots T[1]$

Register of notations.

Symbol	Meaning
α	load factor of a hash table
D	concatenation of the lists $\mathcal{L}_\lambda$ $\forall\lambda$
F_x	factor
λ	leaf
$\mathcal{L}_\lambda$	list of witnesses encountered during a leaf-to-root traversal from λ
L	list storing pairs representing squares
M	hash table size
m	$m \coloneqq \alpha M$ maximal number of elements a hash table can store
n_v	exploration counter of the in-going edge of a suffix tree node v
u, v, w	nodes of a tree
x, y	counting variables for the range $1, \ldots, z$
X, Y	arrays
z	number of LZ77 or LZ78 factors

Register of common variables in Chapter 3.

Symbol	Meaning
β, γ	blocks
$\mathcal{B}$	binary search prefix tree
$\mathcal{C}$	number of characters that have to be compared to decide the sorting of all m suffixes
$\mathfrak{D}$	dictionary of a CFG
η	height at which to cut an HSP tree to form an η-truncated HSP tree
$\mathsf{ET}(T)$	ESP tree of T
$\mathsf{HT}(T)$	HSP tree of T
$\mathcal{L}$	search tree storing created LCE intervals
μ, ν	meta-blocks
M	an instance of dynLCE
m	size of $\mathcal{P}$
$\mathcal{P}$	set of starting positions of suffixes subject to sorting
$\mathsf{SAVL}(\mathit{Suf}(\mathcal{P}))$	suffix AVL tree storing all suffixes of T whose starting positions belong to $\mathcal{P}$
$\mathsf{SLCP}(T, \mathcal{P})$	sparse LCP array of T with respect to $\mathcal{P}$
$\mathsf{SSA}(T, \mathcal{P})$	sparse suffix array of T indexing all suffixes whose starting positions belong to $\mathcal{P}$
$\mathit{Suf}(\mathcal{P})$	the set of all suffixes whose starting positions belong to $\mathcal{P}$
$\mathsf{tHT}_\eta(T)$	η-truncated HSP tree tree of T for a height η
$t_{\mathrm{C}}(\ell)$	time to construct an dynLCE on ℓ characters
$t_{\mathrm{M}}(\ell)$	time to merge two dynLCEs covering ℓ characters
$t_{\mathrm{Q}}(\ell)$	time to query an dynLCE covering ℓ characters
X, Y, Z	substrings of T

Register of common variables in Chapter 4.

Symbol	Meaning
α	$\alpha \in \mathbb{R}$ with $\alpha \geq 1$. $(U_\mathrm{L}, U_\mathrm{R})$ α-gapped $:\Leftrightarrow g + \lvert U_\mathrm{L}\rvert = q \leq \alpha \lvert U_\mathrm{L}\rvert$
β	$\beta \in (0,1] \subset \mathbb{R}$. $(U_\mathrm{L}, U_\mathrm{R})$ β-periodic $:\Leftrightarrow U_\mathrm{L}[1 \mathinner{..} \beta \lvert U_\mathrm{L}\rvert]$ or $U_\mathrm{R}[1 \mathinner{..} \beta \lvert U_\mathrm{R}\rvert]$ periodic in case of gapped repeats or gapped palindromes, respectively
γ	constant with $\gamma \geq 4$ to separate α-gapped repeats/palindromes based on whether their arms are at most or longer than $\gamma \lg n$
$\mathcal{C}_n$	$\{(x,y) \mid 1 \leq y \leq n-1 \text{ and } 1 \leq x \leq n-y\}$
C	subset of $\mathcal{C}_n$
$\mathsf{corr}(i)$	$(i+1 \mod 2)/2$
φ	string isomorphism
g	gap of $(U_\mathrm{L}, U_\mathrm{R})$, i.e., $g \coloneqq \mathsf{b}(U_\mathrm{R}) - \mathsf{e}(U_\mathrm{L}) - 1$
μ, ν	functions
m	combinatoric part: center of the left arm of a gapped palindrome
m	algorithmic part: counting variable for superblocks
$\mathcal{P}_\alpha(T)$	all maximal α-gapped palindromes in T
$\mathcal{P}_\alpha^\beta(T)$	all maximal α-gapped β-periodic palindromes in T
$\overline{\mathcal{P}_\alpha^\beta}(T)$	all maximal α-gapped β-aperiodic palindromes in T
q	arm-period of $(U_\mathrm{L}, U_\mathrm{R})$, i.e., $q \coloneqq \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L})$
R	run
$\mathcal{R}_\alpha(T)$	all maximal α-gapped repeats in T
$\mathcal{R}_\alpha^\beta(T)$	all maximal α-gapped β-periodic repeats in T
$\overline{\mathcal{R}_\alpha^\beta}(T)$	all maximal α-gapped β-aperiodic repeats in T
$(S_\mathrm{L}, S_\mathrm{R})$	a gapped repeat/palindrome contained in the gapped repeat/palindrome $(U_\mathrm{L}, U_\mathrm{R})$
$\widetilde{T}$	block-representation of T
$(U_\mathrm{L}, U_\mathrm{R})$	gapped repeat or gapped palindrome
$\mathsf{weight}(\hat{p})$	weight of a point $\hat{p} \in \mathbb{Z}^2$
$\xi_\mathcal{R}$	$\xi_\mathcal{R}(U_\mathrm{L}, U_\mathrm{R}) \coloneqq (\mathsf{e}(U_\mathrm{L}), \mathsf{b}(U_\mathrm{R}) - \mathsf{b}(U_\mathrm{L}))$
$\xi_\mathcal{P}$	$\xi_\mathcal{P}(U_\mathrm{L}, U_\mathrm{R}) \coloneqq (\lceil(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2\rceil, \lfloor(\mathsf{b}(U_\mathrm{R}) + \mathsf{e}(U_\mathrm{R}))/2\rfloor - \lceil(\mathsf{b}(U_\mathrm{L}) + \mathsf{e}(U_\mathrm{L}))/2\rceil)$
X_m	superblock

Register of common variables in Chapter 5.

List of Identifiers in Chapter 3

While describing the LZ77 and LZ78 factorization algorithms in Sects. 3.4 and 3.7, we used several data structures, among others bit vectors, some with rank or select-support, to achieve the small space bounds. We denote bit vectors with B_α for an α consisting of one or two upper case letters.

For all types of LZ-factorizations we use

- B_{W} marking all witness nodes,
- the array W mapping witness ranks to

  LZ77 text positions, or

  LZ78 factor indices.

In LZ77 we use

- B_{V} marking visited nodes, and

In LZ78 we use

- the array W' mapping LZ nodes to factor indices,
- B_{C} counting n_v of each partially explored node v,
- B_{V} marking suffix tree nodes represented in the LZ trie (their ingoing edges are fully explored),
- B_{LZ} marking explicit LZ nodes, and
- B_{E} marking the edge witnesses.

The algorithms based on the SST additionally use

- B_{T} marking the starting positions of all factors, used also for representing the length of a factor.

We count the number of

- factors by z,
- witnesses by z_{W},
- referencing factors by z_{R}, and
- fresh factors by z_{F}.

LZ77 Output-Streaming with SST, see Sect. 3.4.1

Name	Bits	Rank	Select	In-Place
SA.RMQ	$2n + o(n)$			
B_V				

Common in Sects. 3.4 and 3.7 except for Sect. 3.4.1

Name	Bits	Rank	Select	In-Place
B_W	$n + o(n)$	◯		
B_V				

LZ77 with CST, see Sect. 3.4.2

Name	Bits	(a)	(b)	Rank	Select	In-Place
W	$z \lg n$	◯	◯			

LZ77 with SST In-Place, see Sect. 3.4.3

Name	Bits	(a)	(b)	(c)	(M)	Rank	Select	In-Place
D	$(z_W + z_R) \lg n$			◯				◯
B_D	$n + z_W + z_R$		◯	◯	◯			
B_M	$z_W + z_R$				◯			
B_T	$n + o(n)$	◯	◯	◯	◯		◯	
B_Z	z		◯	◯				

Common for LZ78, see Sect. 3.7.1

Name	Bits	Rank	Select	In-Place
B_C				
B_V				

LZ78 with CST, see Sect. 3.7.2

Name	Bits	(a)	(b)	(b')	Rank	Select	In-Place
B_{LZ}	$z + o(z)$			◯		◯	
B_E	$2n$			◯			
W/W'	$z \lg z$	◯	◯	◯			

LZ78 with SST, Sect. 3.7.3

Name	Bits	(b)	(M)	Rank	Select	In-Place
L	$z \lg n$	◯	◯			◯
W	$z_{\mathrm{W}} \lg n$		◯			◯
B_{L}	$z + z_{\mathrm{W}}$			◯		
B_{T}	$n + o(n)$			◯	◯	

List of data structures with names. The list comprises additional data structures used while computing the LZ factorizations of a text of length n. The letters written in brackets represent a pass (e.g., (a) refers to Pass (a)). The number of bits is omitted if it is exactly n. Circles symbolize that the data structure is used during a pass, or that it is used with a **rank** or **select** structure, or that the data structure is stored in space of X and Y (Column *In-Place*).

Acronyms

BP balanced parentheses

BSPT binary search prefix tree

BWT Burrows-Wheeler transform

CFG context free grammar

CPU central processing unit

CST compressed suffix tree

DNA deoxyribonucleic acid

DFUDS depth-first unary degree sequence

dynLCE dynamic LCE data structure

ESP edit sensitive parsing

HSP hierarchical stable parsing

LCA lowest common ancestor

LCE longest common extension

LCG linear congruential generator

LCP longest common prefix

LCS longest common suffix

LPF longest previous factor

LZ77 Lempel-Ziv-77

LZ78 Lempel-Ziv-78

LZ Lempel-Ziv

LZSS Lempel-Ziv-Storer-Szymanski

LZW Lempel-Ziv-Welch

MAST minimal augmented suffix tree

RAM random-access memory or machine

RNA ribonucleic acid

RMQ range minimum query

SDSL Succinct Data Structure Library

SEDM string edit distance with moves

SLCP sparse longest common prefix array

SSA sparse suffix array

SSD solid-state disk

SST succinct suffix tree

Bibliography

[1] A. Abeliuk, R. Cánovas, and G. Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *JDA*, 2(1):53–86, 2004.

[3] J. A. Adam. *Mathematics in Nature: Modeling Patterns in the Natural World*. Princeton University Press, 2006.

[4] S. Alstrup, T. Husfeldt, and T. Rauhe. Marked ancestor problems. In *Proc. FOCS*, pages 534–543, 1998.

[5] S. Alstrup, G. S. Brodal, and T. Rauhe. Pattern matching in dynamic texts. In *Proc. SODA*, pages 819–828, 2000.

[6] A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119 (2):258–282, 1995.

[7] A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119 (2):258–282, 1995.

[8] A. Andersson and S. Nilsson. A new efficient radix sort. In *Proc. FOCS*, pages 714–721, 1994.

[9] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. Computer and System Sciences*, 18(2):155–193, 1979.

[10] A. Apostolico and S. Lonardi. Compression of biological sequences by greedy off-line textual substitution. In *Proc. DCC*, pages 143–152, 2000.

[11] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996.

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

[13] D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. *Information and Computation*, 209(7):1070–1102, 2011.

[14] D. Arroyuelo, P. Davoodi, and S. R. Satti. Succinct dynamic cardinal trees. *Algorithmica*, 74 (2):742–777, 2016.

[15] D. Arroyuelo, R. Cánovas, G. Navarro, and R. Raman. LZ78 compression in low main memory space. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 38–50, 2017.

[16] J. Arz and J. Fischer. LZ-compressed string dictionaries. In *Proc. DCC*, pages 322–331, 2014.

[17] N. Askitis. Fast and compact hash tables for integer keys. In *Proc. Australasian Computer Science Conference*, volume 91 of *Conferences in Research and Practice in Information Technology*, pages 101–110, 2009.

[18] G. Badkobeh, S. Chairungsee, and M. Crochemore. Hunting redundancies in strings. In *Proc. Developments in Language Theory*, volume 6795 of *LNCS*, pages 1–14, 2011.

[19] H. Bannai. Grammar compression. In *Encyclopedia of Algorithms*, pages 861–866. Springer, 2016.

[20] H. Bannai, S. Inenaga, and M. Takeda. Efficient LZ78 factorization of grammar compressed text. In *Proc. SPIRE*, volume 7608 of *LNCS*, pages 86–98, 2012.

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

[22] H. Bannai, S. Inenaga, and D. Köppl. Computing all distinct squares in linear time for integer alphabets. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 22:1–22:18, 2017.

[23] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. ISAAC*, volume 6507 of *LNCS*, pages 315–326, 2010.

[24] J. D. Barrow. *The Artful Universe*. Clarendon Press, 1995.

[25] J. D. Barrow. *New Theories of Everything: The Quest for Ultimate Explanation*. Oxford University Press, 2007.

[26] P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *JCSC*, 65(1):38–72, 2002.

[27] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. SODA*, pages 2053–2071, 2016.

[28] D. Belazzougui, V. Mäkinen, and D. Valenzuela. Compressed suffix array. In *Encyclopedia of Algorithms*, pages 386–390. Springer, 2016.

[29] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43 (4):275–292, 2005.

[30] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. SODA*, pages 360–369, 1997.

[31] P. Bille, I. L. Gørtz, B. Sach, and H. W. Vildhøj. Time-space tradeoffs for longest common extensions. *JDA*, 25:42–50, 2014.

[32] P. Bille, I. L. Gørtz, M. B. T. Knudsen, M. Lewenstein, and H. W. Vildhøj. Longest common extensions in sublinear space. In *Proc. CPM*, volume 9133 of *LNCS*, pages 65–76, 2015.

[33] P. Bille, J. Fischer, I. L. Gørtz, T. Kopelowitz, B. Sach, and H. W. Vildhøj. Sparse text indexing in small space. *TALG*, 12 (3):39:1–39:19, 2016.

[34] J. R. Black, C. U. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Proc. Workshop on Algorithms and Data Structures*, pages 37–48, 1998.

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

[36] D. Breslauer and G. F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *JDA*, 18:32–48, 2013.

[37] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In *Proc. CPM*, volume 1645 of *LNCS*, pages 134–149, 1999.

[38] G. S. Brodal, R. B. Lyngsø, A. Östlin, and C. N. S. Pedersen. Solving the string statistics problem in time $\mathcal{O}(n \log n)$. In *Proc. ICALP*, volume 2380 of *LNCS*, pages 728–739, 2002.

[39] G. S. Brodal, P. Davoodi, and S. S. Rao. Path minima queries in dynamic weighted trees. In *Proc. WADS*, volume 6844 of *LNCS*, pages 290–301, 2011.

[40] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[41] V. Bush. As we may think. *The Atlantic*, 176(1):101–108, 1945.

[42] L. Carter and M. N. Wegman. Universal classes of hash functions. *JCSC*, 18(2):143–154, 1979.

[43] S. Chairungsee. Searching for gapped palindrome. In *Proc. DEXA*, pages 61–63, 2016.

[44] S. Chairungsee and M. Crochemore. Efficient computing of longest previous reverse factors. In *Proc. Computer Science and Information Technologies*, pages 27–30, 2009.

[45] S. Chairungsee and M. Crochemore. Longest previous non-overlapping factors table computation. In *Proc. Combinatorial Optimization and Applications*, volume 10628 of *LNCS*, pages 483–491, 2017.

[46] S. Chairungsee, T. Butrak, S. Chareonrak, and T. Charuphanthuset. Longest previous non-overlapping factors computation. In *Proc. DEXA*, pages 5–8, 2015.

[47] T. M. Chan, J. I. Munro, and V. Raman. Selection and sorting in the "restore" model. In *Proc. SODA*, pages 995–1004, 2014.

[48] K. Chung, M. Mitzenmacher, and S. P. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory of Computing*, 9:897–945, 2013.

[49] D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[50] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Transactions on Computers*, 33(9):828–834, 1984.

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

[52] C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *IPL*, 75(1-2):9–12, 2000.

[53] R. Cole and R. Hariharan. Dy-

namic LCA queries on trees. *SICOMP*, 34(4):894–923, 2005.

[54] G. Cormode and S. Muthukrishnan. Substring compression problems. In *Proc. SODA*, pages 321–330, 2005.

[55] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *TALG*, 3(1):2:1–2:19, 2007.

[56] M. Crochemore. Recherche linéaire d'un carré dans un mot. *Comptes Rendus des Séances de l'Académie des Sciences. Série I. Mathématique*, 296(18):781–784, 1983.

[57] M. Crochemore. Transducers and repetitions. *TCS*, 45(1):63–86, 1986.

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

[59] M. Crochemore and W. Rytter. Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays. *TCS*, 88(1):59–82, 1991.

[60] M. Crochemore and G. Tischler. Computing longest previous non-overlapping factors. *IPL*, 111(6):291–295, 2011.

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

[62] M. Crochemore, L. Ilie, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. LPF computation revisited. In *Proc. IWOCA*, volume 5874 of *LNCS*, pages 158–169, 2009.

[63] M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *TCS*, 410(50):5227–5235, 2009.

[64] M. Crochemore, C. S. Iliopoulos, M. Kubica, W. Rytter, and T. Walen. Efficient algorithms for three variants of the LPF table. *JDA*, 11:51–61, 2012.

[65] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. Extracting powers and periods in a word from its runs structure. *TCS*, 521:29–41, 2014.

[66] M. Crochemore, R. Kolpakov, and G. Kucherov. Optimal bounds for computing α-gapped repeats. In *Proc. LATA*, volume 9618 of *LNCS*, pages 245–255, 2016.

[67] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[68] A. Deza, F. Franek, and A. Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015.

[69] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, volume 75 of *LIPIcs*, pages 13:1–13:22, 2017.

[70] M. Dumitran, P. Gawrychowski, and F. Manea. Longest gapped repeats and palindromes. *Discrete Mathematics & Theoretical Computer Science*, 19(4):1–32, 2017.

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

[72] H. El-Zein, J. I. Munro, and M. Robertson. Raising permutations to powers in place. In *Proc. ISAAC*, volume 64 of *LIPIcs*, pages 29:1–29:12, 2016.

[73] P. Elias. Universal codeword sets and representations of the integers. *ITIT*, 21(2):194–203, 1975.

[74] S. Evans, J. Hershey, and G. Saulnier. Kolmogorov complexity estimation and analysis. Technical Report 2002GRC177, GE Global Research, 2002.

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

[76] A. Farruggia, P. Ferragina, A. Frangioni, and R. Venturini. Bicriteria data compression. In *Proc. SODA*, pages 1582–1595, 2014.

[77] J. A. Feldman and J. R. Low. Comment on Brent's scatter storage algorithm. *Communications of the ACM*, 16(11):703, 1973.

[78] P. Ferragina and G. Manzini. Indexing compressed text. *JACM*, 52(4):552–581, 2005.

[79] P. Ferragina, I. Nitto, and R. Venturini. On the bit-complexity of Lempel-Ziv compression. *SICOMP*, 42(4):1521–1541, 2013.

[80] N. J. Fine and H. S. Wilf. Uniqueness theorem for periodic functions. *Proc. AMS*, 16:109–114, 1965.

[81] J. Fischer. Wee LCP. *IPL*, 110 (8–9):317–320, 2010.

[82] J. Fischer. Inducing the LCP-array. In *Proc. WADS*, volume 6844 of *LNCS*, pages 374–385, 2011.

[83] J. Fischer. Combined data structure for previous- and next-smaller-values. *TCS*, 412(22): 2451–2456, 2011.

[84] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. *ArXiv*, abs/1302.3347, 2013.

[85] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. CPM*, volume 9133 of *LNCS*, pages 160–171, 2015.

[86] J. Fischer and V. Heun. Space efficient preprocessing schemes for range minimum queries on static arrays. *SICOMP*, 40(2):465–492, 2011.

[87] J. Fischer and D. Köppl. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 191–207, 2017.

[88] J. Fischer, T. Gagie, P. Gawrychowski, and T. Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Proc. ESA*, volume 9294 of *LNCS*, pages 533–544, 2015.

[89] J. Fischer, T. I, and D. Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184, 2015.

[90] J. Fischer, T. I, and D. Köppl.

Deterministic sparse suffix sorting on rewritable texts. In *Proc. LATIN*, volume 9644 of *LNCS*, pages 483–496, 2016.

[91] J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting in the restore model. *ArXiv*, abs/1509.07417v2, 2018.

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

[93] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. Combinatorial Theory, Series A*, 82(1):112–120, 1998.

[94] G. Franceschini and R. Grossi. No sorting? Better searching! *TALG*, 4(1):2:1–2:13, 2008.

[95] G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. ICALP*, volume 4596 of *LNCS*, pages 533–545, 2007.

[96] G. Franceschini and S. Muthukrishnan. Optimal suffix selection. In *Proc. STOC*, pages 328–337, 2007.

[97] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu. Radix sorting with no extra space. In *Proc. ESA*, volume 4698 of *LNCS*, pages 194–205, 2007.

[98] F. Franek, J. Holub, W. F. Smyth, and X. Xiao. Computing quasi suffix arrays. *J. of Automata, Languages and Combinatorics*, 8(4):593–606, 2003.

[99] Y. Fujishige, M. Nakamura, S. Inenaga, H. Bannai, and M. Takeda. Finding gapped palindromes online. In *Proc. IWOCA*, volume 9843 of *LNCS*, pages 191–202, 2016.

[100] S. Fukunaga, Y. Takabatake, T. I, and H. Sakamoto. Online grammar compression for frequent pattern discovery. In *Proc. International Conference on Grammatical Inference*, volume 57 of *Workshop and Conference Proceedings*, pages 93–104, 2016.

[101] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. LATA*, volume 7183 of *LNCS*, pages 240–251, 2012.

[102] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proc. LATIN*, volume 8392 of *LNCS*, pages 731–742, 2014.

[103] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

[104] P. Gawrychowski and T. Kociumaka. Sparse suffix tree construction in optimal time and space. In *Proc. SODA*, pages 425–439, 2017.

[105] P. Gawrychowski, F. Manea, R. Mercas, D. Nowotka, and C. Tiseanu. Finding pseudo-repetitions. In *Proc. STACS*, volume 20 of *LIPIcs*, pages 257–268, 2013.

[106] P. Gawrychowski, F. Manea, and D. Nowotka. Testing generalised freeness of words. In *Proc.*

STACS, volume 25 of *LIPIcs*, pages 337–349, 2014.

[107] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Efficiently finding all maximal α-gapped repeats. In *Proc. STACS*, volume 47 of *LIPIcs*, pages 39:1–39:14, 2016.

[108] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, and F. Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes. *TOCS*, 62(1):162–191, 2018.

[109] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Łącki, and P. Sankowski. Optimal dynamic strings. In *Proc. SODA*, pages 1509–1528, 2018.

[110] S. Gog and E. Ohlebusch. Compressed suffix trees: Efficient computation and storage of lcp-values. *JEA*, 18, 2013.

[111] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, volume 8504 of *LNCS*, pages 326–337, 2014.

[112] A. Goldberg, S. Plotkin, and G. Shannon. Parallel symmetry-breaking in sparse graphs. In *Proc. STOC*, pages 315–324, 1987.

[113] G. H. Gonnet and R. A. Baeza-Yates. An analysis of the Karp-Rabin string matching algorithm. *IPL*, 34(5):271–274, 1990.

[114] K. Goto. Optimal time and space construction of suffix arrays and LCP arrays for integer alphabets. *ArXiv*, abs/1703.01009, 2017.

[115] K. Goto and H. Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. DCC*, pages 133–142, 2013.

[116] K. Goto and H. Bannai. Space efficient linear time Lempel-Ziv factorization for small alphabets. In *Proc. DCC*, pages 163–172, 2014.

[117] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *Proc. CPM*, volume 9133 of *LNCS*, pages 219–230, 2015.

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

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

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

[121] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *JCSC*, 69(4): 525–546, 2004.

[122] T. Hagerup. Sorting and searching on the word RAM. In *Proc. STACS*, volume 1373 of *LNCS*, pages 366–398, 1998.

[123] G. L. Heileman and W. Luo. How caching affects hashing. In *Proc. ALENEX*, pages 141–154, 2005.

[124] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. FOCS*,

pages 251–260. IEEE Computer Society, 2003.

[125] J. Hong and G. Chen. Efficient on-line repetition detection. *TCS*, 407(1-3):554–563, 2008.

[126] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[127] L. C. K. Hui. Color set size problem with application to string matching. In *Proc. CPM*, volume 644 of *LNCS*, pages 230–243, 1992.

[128] T. I. Longest common extensions with recompression. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 18:1–18:15, 2017.

[129] T. I and D. Köppl. Improved upper bounds on all maximal α-gapped repeats and palindromes. *ArXiv*, abs/1802.10355, 2018.

[130] T. I, J. Kärkkäinen, and D. Kempa. Faster sparse suffix sorting. In *Proc. STACS*, volume 25 of *LIPIcs*, pages 386–396, 2014.

[131] T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, and A. Shinohara. Detecting regularities on grammar-compressed strings. *Information and Computation*, 240:74–89, 2015.

[132] L. Ilie. A note on the number of squares in a word. *TCS*, 380(3): 373–376, 2007.

[133] S. Inenaga and H. Bannai. Finding characteristic substrings from compressed texts. *International Journal of Foundations of Computer Science*, 23(2):261–280, 2012.

[134] R. W. Irving and L. Love. The suffix binary search tree and suffix AVL tree. *JDA*, 1(5-6):387–408, 2003.

[135] G. J. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554. IEEE Computer Society, 1989.

[136] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees with applications. *J. Computer and System Sciences*, 78(2):619–631, 2012.

[137] J. Jansson, K. Sadakane, and W.-K. Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 71(4): 969–988, 2015.

[138] N. Jonoska, F. Manea, and S. Seki. A stronger square conjecture on binary words. In *Proc. SOFSEM*, volume 8327 of *LNCS*, pages 339–350, 2014.

[139] J. Jurka. Repeats in genomic DNA: mining and meaning. *Current Opinion in Structural Biology*, 8(3):333–337, 1998.

[140] S. Kanda, K. Morita, and M. Fuketa. Practical implementation of space-efficient dynamic keyword dictionaries. In *Proc. SPIRE*, volume 10508 of *LNCS*, pages 221–233, 2017.

[141] J. Kärkkäinen and D. Kempa. LCP array construction using $\mathcal{O}(sort(n))$ (or less) I/Os. In *Proc. SPIRE*, volume 9954 of *LNCS*, pages 204–217, 2016.

[142] J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

[143] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and

sublinear-size index structures for string matching. In *South American Workshop on String Processing*, pages 141–155, 1996.

[144] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. Computing and Combinatorics*, volume 1090 of *LNCS*, pages 219–230, 1996.

[145] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *JACM*, 53(6): 918–936, 2006.

[146] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Proc. CPM*, volume 5577 of *LNCS*, pages 181–192, 2009.

[147] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. SEA*, volume 7933 of *LNCS*, pages 139–150, 2013.

[148] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM*, volume 7922 of *LNCS*, pages 189–200, 2013.

[149] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[150] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. STOC*, pages 125–136, 1972.

[151] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. CPM*, volume 2089 of *LNCS*, pages 181–192, 2001.

[152] D. Kempa and S. J. Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proc. ALENEX*, pages 103–112, 2013.

[153] Z. Khan, J. S. Bloom, L. Kruglyak, and M. Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25 (13):1609–1616, 2009.

[154] D. Knuth. *Sorting and Searching*, volume III of *The Art of Computer Programming*. Addison-Wesley, 1973.

[155] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *JDA*, 3(2-4):143–156, 2005.

[156] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, and T. Walen. A linear time algorithm for seeds computation. In *Proc. SODA*, pages 1095–1112, 2012.

[157] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. Efficient data structures for the factor periodicity problem. In *Proc. SPIRE*, volume 7608 of *LNCS*, pages 284–294, 2012.

[158] R. Kolpakov. On primary and secondary repetitions in words. *TCS*, 418:71–81, 2012.

[159] R. Kolpakov. On the number of gapped repeats with arbitrary gap. *Theor. Comput. Sci.*, 723: 11–22, 2018.

[160] R. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS*, pages 596–604, 1999.

[161] R. Kolpakov and G. Kucherov. Finding repeats with fixed gap. In *Proc. SPIRE*, pages 162–168, 2000.

[162] R. Kolpakov and G. Kucherov. Finding approximate repetitions under Hamming distance. *TCS*, 1(303):135–156, 2003.

[163] R. Kolpakov and G. Kucherov. Searching for gapped palindromes. *TCS*, 410(51):5365–5373, 2009.

[164] R. Kolpakov, G. Kucherov, and T. A. Starikovskaya. Pattern matching on sparse suffix trees. In *Proc. Data Compression, Communications and Processing*, pages 92–97, 2011.

[165] R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov. Searching of gapped repeats and subrepetitions in a word. *ArXiv*, abs/1309.4055, 2013.

[166] R. Kolpakov, M. Podolskiy, M. Posypkin, and N. Khrapov. Searching of gapped repeats and subrepetitions in a word. *JDA*, 46-47:1–15, 2017.

[167] D. Köppl and K. Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12, 2016.

[168] D. Kosolobov. Faster lightweight Lempel-Ziv parsing. In *Proc. MFCS*, volume 9235 of *LNCS*, pages 432–444, 2015.

[169] S. Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[170] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29 (22):4633–4642, 2001.

[171] D. Lemire. The universality of iterated hashing over variable-length strings. *Discrete Applied Mathematics*, 160(4-5):604–617, 2012.

[172] D. Lemire and O. Kaser. Recursive n-gram hashing is pairwise independent, at best. *Computer Speech & Language*, 24(4): 698–710, 2010.

[173] D. Lemire and O. Kaser. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptographic Engineering*, 6(3): 171–185, 2016.

[174] H. Leung, Z. Peng, and H. Ting. An efficient algorithm for online square detection. *TCS*, 363(1): 69–75, 2006.

[175] M. Li and R. Sleep. An LZ78 based string kernel. In *Proc. Advanced Data Mining and Applications*, volume 3584 of *LNCS*, pages 678–689, 2005.

[176] M. Li and Y. Zhu. Image classification via LZ78 based string kernel: A comparative study. In *Proc. Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 3918 of *LNCS*, pages 704–712, 2006.

[177] Z. Li, J. Li, and H. Huo. Optimal in-place suffix sorting. *ArXiv*, abs/1610.08305, 2016.

[178] M. Lothaire. *Applied Combinatorics on Words*. Number 105 in Encyclopedia of Mathematics. Cambridge University Press, 2005.

[179] H. Luan, X. Du, S. Wang, Y. Ni, and Q. Chen. J^+-tree: A new in-

dex structure in main memory. In *Proc. Database Systems for Advanced Applications*, volume 4443 of *LNCS*, pages 386–397, 2007.

[180] T. Maier and P. Sanders. Dynamic space efficient hashing. In *Proc. ESA*, volume 87 of *LIPIcs*, pages 58:1–58:14, 2017.

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

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

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

[184] F. Manea and S. Seki. Square-density increasing mappings. In *Proc. Combinatorics on Words*, volume 9304 of *LNCS*, pages 160–169, 2015.

[185] G. Marsaglia. Xorshift RNGs. *J. Statistical Software*, 8(14):1–6, 2003.

[186] S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *JDA*, 18:100–112, 2013.

[187] E. M. McCreight. A space-economical suffix tree construction algorithm. *JACM*, 23(2):262–272, 1976.

[188] M. D. McIlroy. A research UNIX reader: Annotated excerpts from the programmer's manual, 1971–1986. Technical Report CSTR 139, AT&T Bell Laboratories, 1987.

[189] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality-tests in polylogarithmic time. In *Proc. SODA*, pages 213–222, 1994.

[190] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. SODA*, pages 408–424, 2017.

[191] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *IPL*, 115(9):655–659, 2015.

[192] G. Navarro. Indexing text using the Ziv-Lempel trie. *JDA*, 2(1): 87–114, 2004.

[193] G. Navarro. Implementing the LZ-index: Theory versus practice. *JEA*, 13(2):2:1.1–2:1.49, 2008.

[194] G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[195] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. *SICOMP*, 43(5):1781–1806, 2014.

[196] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Proc. SEA*, volume 7276 of *LNCS*, pages 295–306, 2012.

[197] G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *TALG*, 10 (3):16:1–16:39, 2014.

[198] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Fully dynamic data structure for LCE queries in compressed space.

In *Proc. MFCS*, volume 58 of *LIPIcs*, pages 72:1–72:15, 2016.

[199] G. Nong. Practical linear-time $\mathcal{O}(1)$-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):15, 2013.

[200] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[201] E. Ohlebusch and S. Gog. Lempel-Ziv factorization revisited. In *Proc. CPM*, volume 6661 of *LNCS*, pages 15–26, 2011.

[202] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proc. SPIRE*, volume 6393 of *LNCS*, pages 322–333, 2010.

[203] D. Okanohara and K. Sadakane. An online algorithm for finding the longest previous factors. In *Proc. ESA*, volume 5193 of *LNCS*, pages 696–707, 2008.

[204] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng. FPGA implementation of GZIP compression and decompression for IDC services. In *Proc. International Conference on Field-Programmable Technology*, pages 265–268, 2010.

[205] R. Pagh. Low redundancy in static dictionaries with constant query time. *SICOMP*, 31(2):353–363, 2001.

[206] A. Poyias and R. Raman. Improved practical compact dynamic tries. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 324–336, 2015.

[207] A. Poyias, S. J. Puglisi, and R. Raman. Compact dynamic rewritable (CDRW) arrays. In *Proc. ALENEX*, pages 109–119, 2017.

[208] N. Prezza. In-place sparse suffix sorting. In *Proc. SODA*, pages 1496–1508, 2018.

[209] S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2): 1–31, 2007.

[210] N. Rahman and R. Raman. Rank and select operations on binary strings. In *Encyclopedia of Algorithms*, pages 748–751. Springer, 2008.

[211] G. G. Richard and A. Case. In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux. *Digital Investigation*, 11, Supplement 2(0):3–12, 2014.

[212] R. M. Robinson. Mersenne and Fermat numbers. *Proc. AMS*, 5 (5):842–846, 1954.

[213] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *JACM*, 28(1):16–24, 1981.

[214] G. Roelofs. *PNG: The Definitive Guide*. O'Reilly, 1999.

[215] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully compressed suffix trees. *TALG*, 7(4):53:1–53:34, 2011.

[216] K. Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proc. SODA*, pages 225–237. ACM/SIAM, 2002.

[217] K. Sadakane. Compressed suf-

fix trees with full functionality. *TOCS*, 41(4):589–607, 2007.

[218] K. Sadakane. For equitable human intellect sharing in an information-oriented society. Interview, National Institute of Informatics, 2010.

[219] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proc. SODA*, pages 1230–1239. ACM/SIAM, 2006.

[220] S. C. Sahinalp and U. Vishkin. Symmetry breaking for suffix tree construction. In *Proc. STOC*, pages 300–309, 1994.

[221] H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Transactions*, 92-D(2):158–165, 2009.

[222] R. V. Samonte and E. E. Eichler. Segmental duplications and the evolution of the primate genome. *Nature Reviews Genetics*, 3(1):65–72, 2002.

[223] G. L. Steele Jr., D. Lea, and C. H. Flood. Fast splittable pseudorandom number generators. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, pages 453–472, 2014.

[224] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *JACM*, 29(4):928–951, 1982.

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

[226] P. Svoboda and A. D. Cara. Hairpin RNA: a secondary structure of primary importance. *Cellular and Molecular Life Sciences*, 63 (7):901–908, 2006.

[227] Y. Takabatake, Y. Tabei, and H. Sakamoto. Improved ESP-index: A practical self-index for highly repetitive texts. In *Proc. SEA*, volume 8504 of *LNCS*, pages 338–350, 2014.

[228] Y. Takabatake, K. Nakashima, T. Kuboyama, Y. Tabei, and H. Sakamoto. siEDM: An efficient string index and search algorithm for edit distance with moves. *Algorithms*, 9(2):26:1–26:18, 2016.

[229] Y. Tanimura, Y. Fujishige, T. I, S. Inenaga, H. Bannai, and M. Takeda. A faster algorithm for computing maximal α-gapped repeats in a string. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 124–136, 2015.

[230] Y. Tanimura, T. I, H. Bannai, S. Inenaga, S. J. Puglisi, and M. Takeda. Deterministic sublinear space LCE data structures with efficient construction. In *Proc. CPM*, volume 54 of *LIPIcs*, pages 1:1–1:10, 2016.

[231] Y. Tanimura, T. Nishimoto, H. Bannai, S. Inenaga, and M. Takeda. Small-space LCE data structure with constant-time queries. In *Proc. MFCS*, volume 83 of *LIPIcs*, pages 10:1–10:15, 2017.

[232] M. Tarailo-Graovac and N. Chen. Using RepeatMasker to identify repetitive elements in genomic sequences. *Current Proto-*

cols in Bioinformatics, 25:4.10.1–4.10.14, 2009.

[233] P. Tchebychev. Mémoire sur les nombres premiers. *Journal de mathématiques pures et appliquées*, 1:366–390, 1852.

[234] B. Trombetta and F. Cruciani. Y chromosome palindromes and gene conversion. *Human Genetics*, 136(5):605–619, May 2017.

[235] Y. Ueki, Diptarama, M. Kurihara, Y. Matsuoka, K. Narisawa, R. Yoshinaka, H. Bannai, S. Inenaga, and A. Shinohara. Longest common subsequence in at least k length order-isomorphic substrings. In *Proc. SOFSEM*, volume 10139 of *LNCS*, pages 363–374, 2017.

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

[237] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *JEA*, 14: 4.2:2–4.2:23, 2009.

[238] M. Vyverman, B. De Baets, V. Fack, and P. Dawyndt. essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29 (6):802–804, 2013.

[239] S. Wandelt and U. Leser. FRESCO: referential compression of highly similar sequences. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(5):1275–1288, 2013.

[240] P. Warburton, J. Giordano, F. Cheung, Y. Gelfand, and G. Benson. Inverted repeat structure of the human genome: The X-chromosome contains a preponderance of large, highly homologous inverted repeats that contain testes genes. *Genome Research*, 14:1861–1869, 2004.

[241] P. Weiner. Linear pattern matching algorithms. In *Proc. of the Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE Computer Society, 1973.

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

[243] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[244] Z. Xu. A minimal periods algorithm with applications. In *Proc. CPM*, volume 6129 of *LNCS*, pages 51–62, 2010.

[245] N. Yoshinaga and M. Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proc. Computing and Combinatorics*, pages 1091–1102, 2014.

[246] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *ITIT*, 23(3): 337–343, 1977.

[247] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *ITIT*, 24 (5):530–536, 1978.

[248] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, 1970.

Abbreviations in the Bibliography

The following abbreviations are used in the bibliography to shorten names of conferences, series, and journals.

Common Abbreviations

Proc. Proceedings (of)

J. Journal (of)

Conferences

ALENEX Algorithm Engineering and Experiments

CPM Annual Symposium on Combinatorial Pattern Matching

DCC Data Compression Conference

DEXA Database and Expert Systems Applications

ESA European Symposium on Algorithms

FOCS Annual Symposium on Foundations of Computer Science

ICALP International Colloquium on Automata, Languages, and Programming

ISAAC International Symposium on Algorithms and Computation

IWOCA International Workshop on Combinatorial Algorithms

LATA International Conference on Language and Automata Theory and Applications

LATIN Latin American Theoretical Informatics Symposium

MFCS International Symposium on Mathematical Foundations of Computer Science

SEA International Symposium on Experimental Algorithms

SODA Symposium on Discrete Algorithms

SOFSEM Conference on Current Trends in Theory and Practice of Informatics

SPIRE String Processing and Information Retrieval

STACS Symposium on Theoretical Aspects of Computer Science

STOC Symposium on the Theory of Computing

WADS Workshop on Algorithms and Data Structures

Series

LIPIcs Leibniz International Proceedings in Informatics

LNCS Lecture Notes in Computer Science

Journals

IPL Information Processing Letters

ITIT IEEE Transactions on Information Theory

JACM Journal of the ACM

JCSC Journal of Computer and System Sciences

JDA Journal of Discrete Algorithms

JEA ACM Journal of Experimental Algorithmics

Proc. AMS Proc. of the American Mathematical Society

SICOMP SIAM Journal on Computing

TALG ACM Transactions on Algorithms

TCS Theoretical Computer Science

TOCS Theory of Computing Systems

Index