Computation over Compressed Strings: Indexing, Sensitivity, and Beyond

赤木, 亨

Computation over Compressed Strings: Indexing, Sensitivity, and Beyond

Toru Akagi

February, 2023

Abstract

A massive amount of information is actively exchanged all over the world as a consequence of rapid innovations in network and sensor technologies. These huge amounts of data may contain valuable information that could assist us to solve problems that are socially and/or economically important, and thus their use has now become a key issue. However, these data are very massive and may surpass memory limits and networking band constraints during processing and analysis. Furthermore, certain kinds of data dynamically change (examples are those in Wikipedia and versioned documents in Git). One of the main ways to solve the concerns of large data structures with such dynamic changes is the development of resilient data structures and compressors that are robust to edit operations.

Since most digital data can be regarded as a sequence of symbols (i.e. a *string*), algorithmics and combinatorics on strings (a.k.a. stringology) can play a central role in solving important problems in information retrieval, data compression, and bioinformatics. This thesis studies *computation over compressed data*, of which the task is to efficiently process compressed data by exploiting useful combinatorial properties of compressed strings. In particular, firstly this thesis deals with the *string indexing problem*, where the task is to build a small (compressed) data structure that supports fast pattern matching queries. Secondly, the thesis considers the *sensitivity* of string compressors which evaluates the perturbation of the compression data size after an edit operation is performed to the input string. Thirdly, the thesis moves on to analyses of string combinatorial objects in the compressed and/or semi-dynamic setting.

String compressors that are mainly discussed in this thesis are grammar-based compressions and run-length encodings (RLE), and a new grammar-based string indexing data structure. The sensitivity of various grammar-based compressors is then analyzed. Minimal absent words (MAWs) are one of the key subjects of string combinatorics, which are not only used directly for compression, but also have several applications in music information retrieval and bioinformatics. As a special case of edit operations, MAWs in the sliding window model that repeatedly deletes one character on the left and adds one character on the right are considered.

The contributions of the thesis are three-fold:

- (A) New grammar-based compressed index that have merits both in theory and in practice;
- (B) Upper and lower bounds on the sensitivity of grammar-based compressors;
- (C) Upper and lower bounds on the number of MAWs in RLE-compressed strings and in the sliding window model.

In (A), we propose a new grammar-compressed indexing structure that is built on grammar-compression based on induced suffix sorting (GCIS). In more details, (A-1) we show that GCIS exhibits a locality sensitive parsing property, which allows us to specify, given a pattern P, certain substrings of P called *cores*, that are similarly parsed in the grammar whenever these occurrences are extensible to occurrences of P. Using this property, we present a new compressed text indexing structure built upon a grammar compression GCIS, which allows us to avoid technical difficulties of GCIS that require long decompression time. (A-2) To examine practical performance of the proposed GCIS-Index, we performed comparative experiments with known methods including the ESP-Index, which is the most relevant our current approach. We compared the index size, construction time and locate time in examination of them, and our GCIS-Index achieved fastest pattern matching in the case of matching long patterns of length 1000 or more. Also, GCIS-Index theoretically takes less computation time than the ESP-Index to locate pattern occurrence after core verification. As a result, we found that GCIS-Index locates patterns with faster than the ESP-Index in some datasets with a large number of pattern occurrences.

In (B), we introduce a new concept “compression sensitivity” which measures how much a single character edit operation (substitution, addition, or deletion) effects the compression ratio in some grammar-based compressors. We prove upper and lower bounds for sensitivity of (B-1) GCIS, which is related to (A), (B-2) LZ-End, a variation of the famous LZ77-compressor, and (B-3) Bisection, which is another grammar-based compression. By applying some ideas from the LZSS upper bounds, (B-4) we also provide non-trivial upper bounds of some grammar compressors, such as the AVL-grammar. As for (B-1), we prove that the size of the GCIS, which is a foundation of the GCIS-Index technology, can increase by a constant factor of 4 after a single-character edit operation to the input string. This means that GCIS has a robust compression ratio against single character edits. We also show a family of strings that actually achieves the worst-case sensitivity of GCIS, which is 4. Regarding (B-2), we show that the size of the LZ-End compression can increase by a factor of 2 after single character editing (substitution, addition, and deletion). A non-trivial upper bound of the sensitivity of LZ-End is also discussed.

In (C), we classify MAWs based on their combinatorial properties and present upper and lower bounds on the number of MAWs in each type. (C-1) We find strict upper and lower bounds for the amount of change in the number of MAWs generated by a shift on the sliding window (deleting one character at the left and adding one character at the right). Furthermore, (C-2) we classify MAWs in RLE-compressed strings into five types and prove upper and lower bounds of their numbers for each type. We also propose a compact representation of MAWs that takes space linear in the RLE-compression size. Finally, (C-3) presents how to construct our compact representation of MAWs stated in (C-2) in $O(m \log m)$ time and $O(m)$ space using some data structures related to the truncated RLE-suffixes, where m is the size of the input RLE-compressed string.

Acknowledgments

First and foremost, I want to thank everyone who helped me with my research at Kyushu University. This work would not have been possible without the great support of Associate Professor Shunsuke Inenaga, my supervisor, and my thesis committee members. Their enormous expertise and depth of knowledge have impacted me throughout my academic study and daily life. While five years have passed since I moved to the current laboratory, my supervisor has provided me with sincere advice on many occasions regarding my research.

I would also express my appreciation to Assistant Professor Dominik Köppl. Even though I had no experimental backgrounds in earlier days, he provided me with a lot of advise and ideas for experiments. Even when things went wrong in the experiment's setting, he worked tirelessly together to fix it. It would have been difficult for me to complete my research without his extraordinary support and encouragement over the past several years.

I would also like to thank Professor Masayuki Takeda, Assistant Professor Yuto Nakashima for their great support on my study. I also appreciate all the support received from Assistant Professor Takuya Mieno. He provided a very valuable contribution in a lot of discussions of my research.

Additionally, I would like to express gratitude to Dr. Mitsuru Funakoshi for his treasured support. His contribution, the final piece to complete my unpolished idea, was always truly influential.

I also express my deep gratitude to Professor Eiji Takimoto, my thesis committee chair, and Professor Hideo Bannai and Associate Professor Kohei Hatano, my thesis committee members.

I also thank all of those in the Department of Informatics, Kyushu University, for their support. My gratitude extends to the technical staffs of our laboratory, who were always with me and took great care of me.

The results in the thesis were partially published in the Proceedings of International Symposium on String Processing and Information Retrieval (SPIRE'21), in the Proceedings of 33rd Annual Symposium on Combinatorial Pattern Matching (CPM'22), and in Theoretical Computer Science vol. 927, and will be published in Information and Computation (already available online). I would like to express my gratitude for all editors, committees, anonymous referees, and publishers.

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

Contents

Chapter 1

Introduction

1.1 Background and motivations

A massive amount of information is actively exchanged all over the world as a consequence of rapid innovations in network and sensor technologies. These huge amounts of data may contain valuable information that could assist us to solve problems that are socially and/or economically important, and thus their use has now become a key issue. However, these data are very massive and may surpass memory limits and networking band constraints during processing and analysis. Furthermore, certain kinds of data dynamically change (examples are those in Wikipedia and versioned documents in Git). One of the main ways to solve the concerns of large data structures with such dynamic changes is the development of resilient data structures and compressors that are robust to edit operations.

Since most digital data can be regarded as a sequence of symbols (i.e. a *string*), algorithmics and combinatorics on strings (a.k.a. stringology) can play a central role in solving important problems in information retrieval, data compression, and bioinformatics. This thesis studies *computation over compressed data*, of which the task is to efficiently process compressed data by exploiting useful combinatorial properties of compressed strings. In particular, firstly this thesis deals with the *string indexing problem*, where the task is to build a small (compressed) data structure that supports fast pattern matching queries. Secondly, the thesis considers the *sensitivity* of string compressors which evaluates the perturbation of the compression data size after an edit operation is performed to the input string. Thirdly, the thesis moves on to analyses of string combinatorial objects in the compressed and/or semi-dynamic setting.

String compressors methods that are mainly discussed in this thesis are grammar-based compressions and run-length encodings (RLE), and a new grammar-based string indexing data structure. The sensitivity of various grammar-based compressors is then analyzed. Minimal absent words (MAWs) are one of the key subjects of string combinatorics, which are not only used directly for compression, but also have several applications in music information retrieval and bioinformatics. As a special case of edit operations, MAWs in the sliding window model that repeatedly deletes one character on the left and adds one character on the right are considered.

The contributions of the thesis are three-fold:

- (A) Development of new compressed grammar index and performance evaluation
- (B) An investigation of the influence of single-character editing on some compression and data structures.
- (C) Upper and lower bounds on the number of MAW on compressed models such as RLE strings, sliding windows, and both of them.

1.2 Grammar indexes

The keyword in the first part (A) of this thesis is a compressed index. Although highly-repetitive texts are perceived as a common problem instance nowadays due to use cases involving a set of DNA sequences within a very narrow taxonomy group or a large number of slightly different versions of documents maintained in revision control systems, this problem is difficult to handle due to the usual large amount of data, but interesting in the sense that the data is highly compressible. Compressed text indexes have become standard tools for tackling this problem when full-text search queries such as locating all occurrences of a pattern are of importance. It is desired for a text index to be a *self-index*, i.e., a data structure that supports queries on the underlying text without storing the text in its plain form. One type of such self indexes are *grammar indexes*, which are an augmentation of the admissible grammar [64] produced by a grammar compressor. Grammar indexes exhibit strong compression ratios for artificial texts or texts with high repetition ratios. Unlike other indexes that perform pattern matching step-wise by character-by-character comparison, some grammar indexes have locality sensitive parsing properties, which allows them to match certain non-terminals of the admissible grammar built upon the pattern with the non-terminals of the text. Such a property helps us, to compute fewer comparisons, and thus speed up pattern matching for particularly long patterns, which could be large gene sequences in a genomic database or source code files in a database maintaining source code. Here, we focus on indexes computing the method $locate(P)$ retrieving the starting positions of all occurrences of a given pattern P in a given text.

Our main contribution is the discovery of a locality sensitive parsing property in grammar produced by the grammar compression by induced sorting (GCIS) [94], which helps us to answer locate with an index built upon GCIS with the following bounds:

Theorem 1.1. *Given a text T of length n, we can compute an indexing data structure on T in $O(n)$ time, which can locate all* occ *occurrences of a given pattern of length m in $O(m \lg |\mathcal{S}| + \mathrm{occ}_C \lg m \lg |\mathcal{S}| + \mathrm{occ})$ time, where $\mathcal{S}$ is the set of characters and non-terminals of the GCIS grammar and* occ_C *is the number of occurrences on the right side of the production rules of the GCIS grammar of a selected core of the pattern. Our index can be constructed in $O(n \lg |\mathcal{S}|)$ time, and uses $O(g)$ words of space, where g is the sum of the right sides of all production rules.*

Similar properties hold for other grammars such as the signature encoding [78], ESP [27], HSP [43], the Rsync parse [46], or the grammar of [22]. A brief review of self-indexes follows.

1.3 Compression sensitivity

The second keyword in the next part (B) of this thesis is string edit operations. As mentioned in previous sections, some data can change dynamically in information systems including Github and Wikipedia. One of the most difficult and important issues is calculating the magnitude of slight changes in the size of the compressed representation of the data. For addressing this problem, we introduce a new notion to quantify efficiency of (lossless) compression algorithms, which is our second target that we call *sensitivity* of compressors.

Let C be a compression algorithm and let $C(T)$ denote the size of the output of C applied to an input text (string) T. Roughly speaking, the sensitivity of C measures how much the compressed size $C(T)$ can change when a single-character-wise edit operation is performed at an arbitrary position in T. Namely, the worst-case *multiplicative sensitivity* of C is defined by

$$\max_{T \in \Sigma^n} \{C(T')/C(T) : \mathsf{ed}(T, T') = 1\},$$

where $\mathsf{ed}(T, T')$ denotes the edit distance between T and T'. This new and natural notion enables one to measure the robustness of compression algorithms in terms of errors and/or dynamic changes occurring in the input string. Such errors and dynamic changes are commonly seen in real-world texts such as DNA sequences and versioned documents.

The so-called highly repetitive sequences, which are strings containing a lot of repeated fragments, are abundant today: Semi-automatically generated strings via M2M communications, and collections of individual genomes of the same/close species are typical examples. By intuition, such highly repetitive sequences should be highly compressible, however, statistical compressors are known to fail to capture repetitiveness in a string [69]. Therefore, other types of compressors, such as dictionary-based, grammar-based, and/or lex-based compressors are often used to compress highly repetitive sequences [72, 100, 70, 57].

Let us recall two examples of well-known compressors: The *run-length Burrows-Wheeler Transform* (*RLBWT*) is one kind of compressor based on the lexicographically sorted rotations of the input string. The number r of equal-character runs in the BWT of a string is known to be very small in practice: Indeed, BWT is used in the bzip2 compression format, and several compressed data structures which support efficient queries have been proposed [48, 5, 90, 91]. The *Lempel-Ziv 78 compression* (*LZ78*) [111] is one of the most fundamental dictionary based compressors that is a core of the gif and tiff compression formats. While LZ78 only allows $\Omega(\sqrt{n})$ compression for any string of length n, its simple structure allows for designing efficient compressed pattern matching algorithms and compressed self-indices (c.f. [63, 50, 51, 81, 39] and references therein).

The recent work by Giuliani et al. [54], however, shows that the number r of runs in the BWT of a string of length n can grow by a multiplicative factor of $\Omega(\log n)$ when a single character is prepended to the input string[1]. It is noteworthy that the family of strings discovered by Giuliani et al. [54] satisfies $r(T) \in O(1)$ and $r(T') \in \Omega(\log n)$, where $r(T)$ and $r(T')$ respectively denote the number of runs in the BWTs of T and T'. The other work by Lagarde and Perifel [71] shows that the size of the dictionary of LZ78, which is equal to the number of factors in the respective LZ78 factorization, can grow by a multiplicative factor of $\Omega(n^{1/4})$, again when a single character is prepended to the input string. Letting the LZ78 dictionary size be z_{78}, this multiplicative increase can also be described as $\Omega(z_{78}^{3/2})$. Lagarde and Perifel call this phenomenon "one-bit catastrophe".

Based on these known results, here we introduce the three following classes of string compressors depending on their sensitivity.

(I) Those whose sensitivity is $O(1)$;

(II) Those whose sensitivity is $\mathrm{polylog}(n)$;

(III) Those whose sensitivity is proportional to n^c with some constant $0 < c \leq 1$.

By generalizing the work of Lagarde and Perifel [71], we say that Class (III) is catastrophic in terms of the sensitivity. Class (II) may not be catastrophic but the change in the compression size

[1]If the string ends with a unique symbol \$, then the number r of runs in the BWT increases additively by at most 2 after a character is prepended to the string. Giuliani et al. [54] showed that this is not the case without \$.

can still be quite large just for a mere single character edit operation to the input string. Class (I) is the most robust against one-character edit operations among the three classes. Recall that LZ78 z_{78} belongs to Class (III). As for RLBWT, the work of Giuliani et al. [54] showed only a lower bound $\Omega(\log n)$, however, the multiplicative sensitivity of RLBWT r turns out to be $O(\log r \log n)$, which implies that r belongs to Class (II) [1].[2].

The *LZ77* compression [110], which is the greedy parsing of the input string T where each factor of length more than one refers to a previous occurrence to its left, is the most important dictionary-based compressor both in theory and in practice. The LZ77 compression without self-references (resp. with self-references) can achieve $O(\log n)$ compression (resp. $O(1)$ compression) in the best case as opposed to the $\Omega(\sqrt{n})$ compression by the LZ78 counterpart, and the LZ77 compression is a core of common lossless compression formats including gzip, zip, and png. In addition, its famous version called *LZSS* (Lempel-Ziv-Storer-Szymanski) [101], has numerous applications in string processing, including finding repetitions [29, 67, 56, 7], approximation of the smallest grammar-based compression [98, 21], and compressed self-indexing [12, 13, 83, 10], just to mentioned a few.

In this thesis, we show that such a catastrophe *never happens* with GCIS, which is the grammar compression used in our GCIS-Index we mentioned above. We show that for all types of edit operations, the multiplicative sensitivity of the size of the GCIS grammar is at most 4. Furthermore, that there exist strings for which the multiplicative sensitivity is also 4 with all types of edit operations. We also consider the *smallest bidirectional scheme* [101] that is a generalization of the LZ family where each factor can refer to its other occurrence to its left or right. It is shown that for all types of edit operations, the multiplicative sensitivity of the size b of the smallest bidirectional scheme is at most 2, and that there exist strings for which the multiplicative sensitivity of b is 2 with insertions and substitutions, and it is 1.5 with deletions.

The *smallest grammar* problem [21] is a famous NP-hard problem that asks to compute a grammar of the smallest size g^* that derives only the input string. We show that the multiplicative sensitivity of the smallest grammar size g^* is at most 2. We also present non-trivial upper and lower bounds for the multiplicative sensitivity for the size z_{End} of the LZ-End compressor [69].

Moreover, we consider the multiplicative sensitivity of other compressors and repetitiveness measures including Bisection [87], GCIS [94, 93], α-balanced grammars [21], AVL-grammars [98], and Recompression [59]. Table 1.1 summarizes our results on the multiplicative

[2]This $O(\log r \log n)$ upper bound for the sensitivity of r follows from our result on the sensitivity of δ and the Lemma 2.1 of [1], and from the known results between r and δ [60, 66].

sensitivity of the string compressors and repetitiveness measures.

In addition to the aforementioned multiplicative sensitivity, we also introduce the worst-case *additive sensitivity*, which is defined by

$$\max_{T \in \Sigma^n} \{C(T') - C(T) : \mathsf{ed}(T, T') = 1\},$$

for all the string compressors/repetitiveness measures C dealt with in this thesis. We remark that the additive sensitivity allows one to observe and evaluate more details in the changes of the output sizes, as summarized in Table 1.2. Studying the additive sensitivities of string compressors is motivated by an approximation of the Kolmogorov complexity. Let $K(T)$ denote the Kolmogorov complexity of string T, that is the length of a shortest program that produces T. While $K(T)$ is known to be uncomputable, the additive sensitivity $K(T') - K(T)$ for deletions is at most $O(\log n)$ *bits*, since it suffices to add "Delete the ith character $T[i]$ from T." at the end of the program. Similarly, the additive sensitivity of K for insertions and substitutions is at most $O(\log n + \log \sigma)$ *bits*, where σ is the alphabet size. Therefore, a "good approximation" of the Kolmogorov complexity K should have small additive sensitivity.

Table 1.1: Multiplicative sensitivity of the string compressors and string repetitiveness measures studied in this thesis and in the literature, where n is the input string length and Σ is the alphabet. In the table "sr" stands for "with self-references". The upper bounds marked with "†" are obtained by applying known results [61, 66, 60, 69, 62, 21, 98, 59] and our results on the sensitivity of the substring complexity δ or the smallest grammar g^* to Lemma 2.1. This thesis excludes the boundaries denoted by "§", the published version of this work [1] has detailed proofs of the results.

compressor/repetitiveness measure	edit type	upper bound	lower bound
Substring Complexity δ	ins./subst.	$2^{§}$	$2^{§}$
	deletion	$1.5^{§}$	$1.5^{§}$
Smallest String Attractor γ	all	$O(\log n)^{†§}$	$2^{§}$
RLBWT r	insertion	$O(\log n \log r)^{†§}$	$\Omega(\log n)$ [54]
	del./subst.		-
Bidirectional Scheme b	ins./subst.	$2^{§}$	$2^{§}$
	deletion	$2^{§}$	$1.5^{§}$
LZ77 z_{77} LZ77sr $z_{77\mathrm{sr}}$	all	$2^{§}$	$2^{§}$
LZSS z_{SS} LZSSsr z_{SSsr}	del./subst.	$3^{§}$	$3^{§}$
	insertion	$2^{§}$	$2^{§}$
LZ78 z_{78}	insertion	$O((n/\log n)^{\frac{2}{3}})^{†}$	$\Omega(n^{\frac{1}{4}})$ [71]
	del./subst.		$\Omega(n^{\frac{1}{4}})^{§}$
LZ-End z_{End}	all	$O(\log^2(n/\delta))^{†}$	2
Smallest grammar g^*	all	2	-
Repair g_{rpair}	all	$O((n/\log n)^{\frac{2}{3}})^{†}$	-
Longest match g_{long}			
Greedy g_{grdy}			
Sequential g_{seq}	all	$O((n/\log n)^{\frac{3}{4}})^{†}$	-
α-balanced grammar g_α	all	$O(\log(n/g^*))^{†}$	-
AVL grammar g_{avl}			
Simple g_{simple}			
Bisection g_{bsc}	substitution	2	2
	ins./del.	$\sigma + 1$	σ
GCIS g_{is}	all	4	4

Table 1.2: Additive sensitivity of the string compressors and string repetitiveness measures studied in this thesis, where n is the input string length and σ is the alphabet size. Some upper/lower bounds are described in terms of both the measure and n. In the table "sr" stands for "with self-references". The upper bounds marked with "†" are obtained by applying known results [61, 66, 60, 69, 62, 21, 98, 59] and our results on the sensitivity of the substring complexity δ or the smallest grammar g^* to Lemma 2.1. This thesis excludes the boundaries denoted by "§". The published version of this work [1] has detailed proofs of the results.

compressor/ repetitiveness measure	edit type	upper bound		lower bound	
Substring Complexity δ	all	$1^{\S}$		$1^{\S}$	
Smallest String Attractor γ	all	$O(\delta \log n)^{\dagger\S}$		$\gamma - 3^{\S}$	$\Omega(\sqrt{n})^{\S}$
RLBWT r	insertion	$O(r \log n \log r)^{\dagger\S}$		-	$\Omega(\log n)$ [54]
	del./subst.				-
Bidirectional Scheme b	all	$b + 2^{\S}$		$b/2 - 3^{\S}$	$\Omega(\sqrt{n})^{\S}$
LZ77 z_{77}	subst./ins.	$z_{77} - 1^{\S}$		$z_{77} - 1^{\S}$	$\Omega(\sqrt{n})^{\S}$
	deletion	$z_{77} - 2^{\S}$		$z_{77} - 2^{\S}$	
LZ77sr z_{77}	subst./ins.	$z_{77\mathrm{sr}}{}^{\S}$		$z_{77\mathrm{sr}}{}^{\S}$	$\Omega(\sqrt{n})^{\S}$
	deletion			$z_{77\mathrm{sr}} - 2^{\S}$	
LZSS z_{SS}	del./subst.	$2z_{\mathrm{SS}} - 2^{\S}$		$2z_{\mathrm{SS}} - \Theta(\sqrt{z_{\mathrm{SS}}})^{\S}$	$\Omega(\sqrt{n})^{\S}$
	insertion	$z_{\mathrm{SS}}{}^{\S}$		$z_{\mathrm{SS}} - \Theta(\sqrt{z_{\mathrm{SS}}})^{\S}$	
LZSSsr z_{SSsr}	del./subst.	$2z_{\mathrm{SSsr}}{}^{\S}$		$2z_{\mathrm{SSsr}} - \Theta(\sqrt{z_{\mathrm{SSsr}}})^{\S}$	$\Omega(\sqrt{n})^{\S}$
	insertion	$z_{\mathrm{SSsr}} + 1^{\S}$		$z_{\mathrm{SSsr}} - \Theta(\sqrt{z_{\mathrm{SSsr}}})^{\S}$	
LZ78 z_{78}	insertion	$O(g^* \cdot (n/\log n)^{\frac{2}{3}})^{\dagger\S}$		$\Omega((z_{78})^{\frac{3}{2}})$ [71]	$\Omega(n/\log n)$ [71]
	del./subst.			$\Omega((z_{78})^{\frac{3}{2}})^{\S}$	$\Omega(n^{\frac{3}{4}})^{\S}$
LZ-End z_{End}	all	$O(z_{\mathrm{End}} \log^2(n/\delta))^{\dagger}$		$z_{\mathrm{End}} - \Theta(\sqrt{z_{\mathrm{End}}})$	$\Omega(\sqrt{n})$
Smallest grammar g^*	all	g^*		-	
Repair g_{rpair}	all	$O(g^* \cdot (n/\log n)^{\frac{2}{3}})^{\dagger}$		-	
Longest match g_{long}					
Greedy g_{grdy}					
Sequential g_{seq}	all	$O(g^* \cdot (n/\log n)^{\frac{3}{4}})^{\dagger}$		-	
α-balanced grammar g_{α}	all	$O(g^* \log(n/g^*))^{\dagger}$		-	
AVL grammar g_{avl}					
Simple g_{simple}					
Bisection g_{bsc}	substitution	g_{bsc}	$\lceil \log_2 n \rceil$	$g_{\mathrm{bsc}} - 4$	$2\log_2 n - 4$
	ins./del.	σg_{bsc}		$\Omega(\sigma g_{\mathrm{bsc}})$	$\Omega(\sigma^2 \log_2(n/\sigma))$
GCIS g_{is}	all	$3g_{\mathrm{is}}$		$3g_{\mathrm{is}} - 26$	$\Omega(n)$

1.4 Minimal absent words (MAWs)

Edit operations affect not only compression sensitivity, but also other properties of strings, such as palindromes, minimal unique substrings, and so on. Most, if not all, of them are useful to analyze string data, and therefore, the number of such characteristic features and the amount of change are also of significance. One of such characteristic features we focus on is *minimal absent words* which is the last keyword in the third part (C) of this thesis.

We say that a string s *occurs* in another string T if s is a substring of T. A non-empty string w is said to be a *minimal absent word* (an *MAW*) for a string T if w does not occur in T but all proper substrings of w occur in T. Note that by definition a string of length 1 (namely a character) which does not occur in T is also an MAW for T. On the other hand, any MAW for T of length at least 2 can be represented as aub, where a and b are single characters and u is a (possibly empty) string, such that both au and ub occur in T. For example, let $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ be the alphabet. Then, the set of MAWs for string $w = \mathtt{abaab}$ is $\{\mathtt{aaa}, \mathtt{aaba}, \mathtt{bab}, \mathtt{bb}, \mathtt{c}\}$.

Applications of (minimal) absent words include phylogeny [18], data compression [32, 33], musical information retrieval [28], and bioinformatics [3, 95, 68, 19].

In this thesis, we first analyze the amount of MAWs for the sliding window model, and second, we propose the efficient representation of MAWs for run-length encodings (RLE).

1.4.1 MAWs for sliding window

We follow the recent line of research on *MAWs for the sliding window model*, which was initiated by Crochemore et al. [30]. In this model, the goal is to compute or analyze $\mathsf{MAW}(T[i..i + d - 1])$, as i is incremented, each time by 1, from 1 to $n - d + 1$. For intuition, consider sliding a length-d window on T from left to right.

Crochemore et al. [30] presented a suffix-tree based algorithm that maintains the set of all MAWs for a sliding window in $O(\sigma n)$ time using $O(\sigma d)$ working space. Crochemore et al. [30] also showed how their algorithm can be applied to approximate pattern matching under the *length weighted index* (*LWI*) metric [18].

How efficiently their system works is heavily dependent on combinatorial properties of MAWs for the sliding window. In particular, Crochemore et al. [30] studied the number of MAWs to be added/deleted when the current window is shifted to the right by one character. As was done in [30], for ease of discussion let us separately consider

- adding a new character $T[i + d]$ to the current window $T[i..i + d - 1]$ of length d which forms $T[i..i+d]$, and

- deleting the leftmost character $T[i-1]$ from the current window $T[i-1..i+d-1]$ which forms $T[i..i+d-1]$ of length d.

We remark that these two operations are symmetric.

Crochemore et al. [30] considered how many MAWs can change before and after the window has been shifted by one position, and showed that

$$|\mathsf{MAW}(T[i..i+d]) \triangle \mathsf{MAW}(T[i..i+d-1])| \leq (s_i - s_\alpha)(\sigma - 1) + \sigma + 1,$$
$$|\mathsf{MAW}(T[i-1..i+d-1]) \triangle \mathsf{MAW}(T[i..i+d-1])| \leq (p_i - p_\beta)(\sigma - 1) + \sigma + 1,$$

where $\triangle$ denotes the symmetric difference and

- s_i is the length of the longest repeating suffix of $T[i..i+d-1]$,
- s_α is the length of the longest suffix of $T[i..i+d-1]$ having an internal occurrence immediately followed by $\alpha = T[i+d]$,
- p_i is that of the longest repeating prefix of $T[i..i+d-1]$, and
- p_β is the length of the longest prefix of $T[i..i+d-1]$ having an internal occurrence immediately preceded by $\beta = T[i-1]$.

Since both $s_i - s_\alpha$ and $p_i - p_\beta$ can be at most $d-1$ in the worst case, the asymptotic bounds for the numbers of changes in the set of MAWs obtained by Crochemore et al. [30] are:

$$\begin{aligned} |\mathsf{MAW}(T[i..i+d]) \triangle \mathsf{MAW}(T[i..i+d-1])| &\in O(\sigma d), \\ |\mathsf{MAW}(T[i-1..i+d-1]) \triangle \mathsf{MAW}(T[i..i+d-1])| &\in O(\sigma d). \end{aligned} \tag{1.1}$$

Crochemore et al. [30] also considered the *total changes* in the set of MAWs for every sliding window over the string T, and showed that

$$\sum_{i=1}^{n-d} \left(|\mathsf{MAW}(T[i..i+d-1]) \triangle \mathsf{MAW}(T[i+1..i+d])| \right) \in O(\sigma n). \tag{1.2}$$

The goal of this thesis is to give more rigorous analyses on the number of MAWs for the sliding window model. This study is well motivated since revealing more combinatorial insights to the sets of MAWs for the sliding windows can lead to more efficient algorithms for computing them.

In this thesis, we first give the following upper bounds:

$$\begin{aligned} |\mathsf{MAW}(T[i..i+d])\triangle\mathsf{MAW}(T[i..i+d-1])| &\leq d+\sigma_{i,i+d-1}+1, \\ |\mathsf{MAW}(T[i-1..i+d-1])\triangle\mathsf{MAW}(T[i..i+d-1])| &\leq d+\sigma_{i,i+d-1}+1, \end{aligned} \tag{1.3}$$

where $\sigma_{x,y}$ is the number of distinct characters in $T[x,y]$. We then show that our new upper bounds in (1.3) are *tight* by showing a family of strings achieving these bounds.

Since $\sigma_{i,i+d-1} \leq d$ always holds, we immediately obtain new asymptotic upper bounds

$$\begin{aligned} |\mathsf{MAW}(T[i..i+d])\triangle\mathsf{MAW}(T[i..i+d-1])| &\in O(d), \\ |\mathsf{MAW}(T[i-1..i+d-1])\triangle\mathsf{MAW}(T[i..i+d-1])| &\in O(d). \end{aligned} \tag{1.4}$$

Our new upper bounds in (1.4) improve Crochemore et al.'s upper bounds in (1.1) for any alphabet of size $\sigma \in \omega(1)$. Our upper bounds in (1.4) are also *tight* as there exists a family of strings achieving the matching lower bounds $\Omega(d)$.

In this thesis, we also present a new upper bound for the total changes of MAWs:

$$\sum_{i=1}^{n-d}\big(|\mathsf{MAW}(T[i..i+d-1])\triangle\mathsf{MAW}(T[i+1..i+d])|\big) \in O(\min\{\sigma,d\}n) \tag{1.5}$$

which improves the previous bound $O(\sigma n)$ in (1.2). We then show that this new upper bound in (1.5) is also *tight*.

All of our new bounds aforementioned are tight for any alphabet of size $\sigma_{i,i+d-1} \geq 3$. We further explore the case of binary alphabets with $\sigma_{i,i+d-1} = 2$, and show that there exist even tighter bounds in the binary case. Namely, for $\sigma_{i,i+d-1} = 2$, we prove that

$$\begin{aligned} |\mathsf{MAW}(T[i..i+d])\triangle\mathsf{MAW}(T[i..i+d-1])| &\leq \max\{3,d\}, \\ |\mathsf{MAW}(T[i-1..i+d-1])\triangle\mathsf{MAW}(T[i..i+d-1])| &\leq \max\{3,d\}. \end{aligned} \tag{1.6}$$

We remark that plugging $\sigma_{i,i+d-1} = 2$ into (1.3) for the general case only gives $d+\sigma_{i,i+d-1}+1 = d+3$, which is larger than $\max\{3,d\}$ in (1.6). We consider the case $\sigma_{i,i+d-1} \geq d$ in Lemmas 3.10 and 3.11. We also show that the upper bounds $\max\{3,d\}$ in (1.6) are *tight* by giving matching lower bounds with a family of binary strings.

A part of the results reported in this article appeared in a preliminary version of this thesis [79]. In addition, this present article considers the case of binary alphabets and presents tight upper and lower bounds for this case.

1.4.2 Computing MAWs in RLE-compressed strings

In this thesis, we initiate the study of computing MAWs for *compressed* strings. As the first step of this line of research, we consider strings which are compactly represented by *run-length* *encoding* (*RLE*). Let m be the size of the RLE of an input string T. We first categorize the elements of $\mathsf{MAW}(T)$ into five disjoint subsets $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, $\mathcal{M}_4$, and $\mathcal{M}_5$, by considering how the MAWs can be related to the boundaries of maximal character runs in T (Section 2.3). In Section 3.2.2 and Section 3.2.1, we present matching upper bounds and lower bounds for their sizes $|\mathcal{M}_i|$ $(i = 1, 2, 4, 5)$ in terms of the RLE size m or the number σ'_T of distinct characters occurring in T. Notice that $\sigma'_T \leq m$ always holds. The exception is $\mathcal{M}_3$, which can contain $\Omega(n)$ MAWs regardless of the RLE size m. Still, in Section 3.2.4 we propose our RLE-compressed $O(m)$-space data structure that can enumerate all MAWs for T in output-sensitive $O(|\mathsf{MAW}(T)|)$ time. Since $m \leq n$ always holds, our result is an improvement over Crochemore et al.'s and Fujishige et al.'s results both of which require $O(n)$ space to store representations of all MAWs. Charalampopoulos et al. [20] showed how one can use *extended bispecial factors* of T to represent all MAWs for T in $O(n)$ space, and to output all MAWs in optimal $O(|\mathsf{MAW}(T)|)$ time upon a query. While the way how we characterize the MAWs may be seen as the RLE version of their method based on the extended bispecial factors, our $O(m)$-space data structure cannot be obtained by a straightforward extension from [20], since there exists a family of strings over a constant-size alphabet for which the RLE-size is $m \in O(1)$ but $|\mathsf{MAW}(T)| \in \Omega(n)$. We note that, by the use of *truncated RLE suffix arrays* [104], our $O(m)$-space data structure can be built in $O(m \log m)$ time with $O(m)$ working space.

1.5 Related work

1.5.1 Compressed string data structures

A compressed string data structure is built on a compressed representation of the string and supports efficient queries such as pattern matching and substring extraction within compressed space. Since the string compressors and string repetitiveness measures that we deal with in this thesis are models for highly repetitive strings, we mention some compressed string indexing structures for highly repetitive sequences below.

The Block tree of a string of length n uses $O(z_{\mathrm{SS}} \log(n/z_{\mathrm{SS}}))$ words of space and supports random access queries in $O(\log(n/z_{\mathrm{SS}}))$ time. Navarro [83] proposed an LZ-based indexing structure that uses $O(z_{\mathrm{SS}} \log(n/z_{\mathrm{SS}}))$ words of space and counts the number of occurrences of a query pattern in the text string in $O(m \log^{2+\epsilon} n)$ time, where m is the length of the pattern and $\epsilon > 0$ is any constant. An $O(\log n)$-time longest common extension (LCE) data structure that takes $O(z_{\mathrm{SS}} \log(n/z_{\mathrm{SS}}))$ space and is based on Recompression [59] was proposed by I [58].

Nishimoto et al. [89] presented a dynamic $O(\min\{z_{\mathrm{SS}} \log n \log^* n, n\})$-space compressed data structure that supports pattern matching and substring insertions/deletions in $O(m \cdot \mathrm{polylog}(n))$ time, where m is the length of the pattern/substring.

Kociumaka et al. [66] proposed a compressed indexing structure that uses $O(\delta \log(n/\delta))$ words of space, performs random access in $O(\log(n/\delta))$ time, and finds the occ occurrences of a given pattern in $O(m \log n + occ \log^{\epsilon} n)$ time. Very recently, Kociumaka et al. [65] proposed an improved data structure of $O(\delta \log(n/\delta))$-space that supports pattern matching queries in $O(m + (occ + 1) \log^{\epsilon} n)$ time. Two independent compressed indexing structures, which are based on grammar compression called GCIS (Grammar Compression by Induced Sorting) [94] have been proposed [2, 35]. Our constant upper bounds on the multiplicative sensitivity for z_{SS}, δ, and g_{is} imply that the aforementioned compressed data structures retain their asymptotic space complexity even after one-character edit operation at an arbitrary position, though they may incur a certain amount of structural changes.

The r-index [48], the refined r-index [5], and the OptBWTR [90] are efficient indexing structures which are built on the RLBWT and use $O(r)$ words of space. The result by Giuliani et al. [54], which uses a family of strings of length n with $r \in O(1)$, shows that the space complexity of these indexing structures can grow from $O(1)$ words of space to $O(\log n)$ words of space, after appending a character to the string. In turn, our upper bound for the sensitivity of r implies that after a one-character edit operation, the space usage of these indexing structures is bounded by $O(r \log r \log n)$ for any string of length n.

There also exist compressed data structures based on other string compressors and/or repetitiveness measures: Kempa and Prezza [61] presented an $O(\gamma\tau \log_\tau(n/\gamma))$-space data structure that allows for extracting substrings of length-ℓ in $O(\log_\tau(n/\gamma) + \ell \log(\sigma)/\omega)$ time, where $\tau \geq 2$ is an integer parameter, σ is the alphabet size, and ω is the machine-word size in the RAM model. Navarro and Prezza [85] gave a data structure of size $O(\gamma \log(n/\gamma))$ that supports pattern matching queries in $O(m \log n + occ \log^{\epsilon} n)$ time. Christiansen et al. [22] introduced a compressed indexing structure that occupies $O(\gamma \log(n/\gamma) \log^{\epsilon} n)$ space and finds all the occ pattern occurrences in optimal $O(m + occ)$ time (for other trade-offs between the space and the query time are also reported, see [22]). Gawrychowski et al. [53] presented a data structure for maintaining a dynamic set of strings, which is based on Recompression by Jeż [59]. Kempa and Saha [62] developed a compressed data structure that occupies $O(z_{\mathrm{End}})$ space and supports random access and LCE queries in $O(\mathrm{polylog}(n))$ time. A compressed indexing structure that can be built directly from the LZ77-compressed text is also known [60, 60]. For other compressed string indexing structures, see this survey [84].

Table 1.3: Needed space (in words) and query time for protect locatefor a pattern of length m regarding the grammar indexes of 1.5.2. Here, n is the length of T, z is the number of LZ77 [110] phrases of T, γ is the size of the smallest string attractor [61] of T, g_{Lyn} is the size of the Lyndon SLP of T, $\hat{g}$ is the size of a given admissible grammar, $\epsilon > 0$ is a constant, m is the length of a pattern P, and occ is the number of occurrences of P in T.

Index	Space	Locate Time
[25]	$O(\hat{g})$	$O(m^2 \lg \lg_{\hat{g}} n + (m + \mathrm{occ}) \lg \hat{g})$
[45]	$O(\hat{g} + z \lg \lg z)$	$O(m^2 + (m + \mathrm{occ}) \lg \lg n)$
[22]	$O(\gamma \lg(n/\gamma))$	$O(m + \lg^{\epsilon} \gamma + \mathrm{occ} \lg^{\epsilon}(\gamma \lg(n/\gamma)))$
[22]	$O(\gamma \lg(n/\gamma) \lg^{\epsilon}(\gamma \lg(n/\gamma)))$	$O(m + \mathrm{occ})$
[105]	$O(g_{\mathrm{Lyn}})$	$O(m + \lg m \lg n + \mathrm{occ} \lg g_{\mathrm{Lyn}})$

1.5.2 Grammar indexes

One area of research in this field is devoted to grammar compression since a grammar can eliminate repetitions while exhibiting powerful query abilities. A grammar index is a *self-index*, i.e., a data structure that supports queries on the underlying text without storing the text in its plain form.

Regarding indexing a general grammar for answering locate, the first work we are aware of is due to [24] who studied indices built upon so-called *straight-line programs (SLPs).* An SLP is a context-free grammar representing a single string in the Chomsky normal form. This result got improved [25, 45, 26]. We present their complexity bounds in Table 1.3.

Other research focused on particular types of grammar, such as the ESP-index [77, 103, 102], an universal index [23] based on Re-Pair [72] and the Lempel–Ziv-77 parsing [110], a dynamic index [89] based on signature encoding [78], the Lyndon SLP [105], or the grammar index of [22]. We show the needed space and query time of these grammars in Table 1.3.

For the experiments, we will additionally have a look to other self-indexes capable of locate-queries. In particular, we study Burrows–Wheeler-transform (BWT) [17]-based approaches, namely the FM-index [41] and the r-index [47], the LZ-index [82, 23] and the RLZ-index [97].

Finally, GCIS has other interesting properties besides being locality sensitive. Nunes et al. showed how to compute the suffix array and the longest-common-prefix array from GCIS during a decompression step restoring the original text [93]. Recently, Díaz-Domínguez and

Navarro proposed an approach to compute the BWT directly from the GCIS grammar [34].

1.5.3 String monotonicity

A string repetitiveness measure C is called *monotone* if, for any string T of length n, $C(T') \leq C(T)$ holds with any of its prefixes $T' = T[1..i]$ and suffixes $T' = T[j..n]$ [66]. Kociumaka et al [66] pointed out that δ is monotone, and posed a question whether γ or the size b of the smallest bidirectional macro scheme [101] are monotone. This monotonicity for C can be seen as a special and extended case of our sensitivity for deletions, namely, if we restrict T' to be the string obtained by deleting either the first or the last character from T, then it is equivalent to asking whether $\max_{T \in \Sigma}\{C(T')/C(T) : T' \in \{T[1..n-1], T' = T'[2..n]\}\} \leq 1$. Mantaci et al. [76] proved that γ is not monotone, by showing a family of strings T such that $\gamma(T) = 2$ and $\gamma(T') = 3$ with $T' = T[1..n-1]$, which immediately leads to a lower bound $3/2 = 1.5$ for the multiplicative sensitivity of γ. Bannai et al. present a new lower bound for the multiplicative sensitivity of γ, which is 2.5 [6] . Mitsuya et al. [80] considered the monotonicity of LZ77 without self-references z_{77} presented a family of strings T for which $z_{77}(T')/z_{77}(T) \approx 4/3$ with $T' = [2..n]$. Again, our matching upper and lower bounds for the multiplicative sensitivity of z_{77}, which are both 2, improve this $4/3$ bound.

1.5.4 Comparison to sensitivity of other algorithms

The notion of the sensitivity of (general) algorithms was first introduced by Varma and Yoshida [106]. They studied the *average* sensitivity of well-known graph algorithms, and presented interesting lower and upper bounds on the expected number of changes in the output of an algorithm A, when a randomly chosen edge is deleted from the input graph G. The worst-case sensitivity of a graph algorithm for edge-deletions and vertex-deletions was considered by Yoshida and Zhou [109].

As opposed to these existing work on the sensitivity of graph algorithms, our notion of the sensitivity of string compressors focuses on the *size* of their compressed outputs rather than the perturbation of their structural changes. This is because the primary task of data compression is to represent the input data with as little memory as possible, and the structural changes of the compressed outputs can be of secondary importance.

We remark that most instances of Σ^n are not compressible, or in other words, a randomly chosen string T from Σ^n is not compressible. Such a string T does not become highly compressible just after a one-character edit operation, and hence $C(T)$ and $C(T')$ are expected to be almost the same. Therefore, considering the average sensitivity of string compressors and repetitiveness measures does not seem worth discussing, and this is the reason why we focus on the worst-case sensitivity of string compressors and repetitiveness measures.

Still, our notion permits one to evaluate the worst-case size changes of several known *compressed string data structures* in the dynamic setting, as will be discussed in the following subsection.

1.5.5 Algorithms for MAWs computation

Given the above-mentioned motivations of MAWs, finding MAWs from a given string has been an important and interesting string algorithmic problem and several nice solutions have been proposed. The first non-trivial algorithm, which was given by Crochemore et al. [31], finds the set $\mathsf{MAW}(T)$ of all MAWs for a given string T of length n over an alphabet of size σ in $\Theta(\sigma n)$ time with $O(n)$ working space. Since $|\mathsf{MAW}(T)| \in O(\sigma n)$ for any string T of length n and $|\mathsf{MAW}(S)| \in \Omega(\sigma n)$ for some string S of length n [31], Crochemore et al.'s algorithm [31] runs in optimal time in the worst case. Later, Fujishige et al. [44] presented an improved data structure of $O(n)$ space, which can report all MAWs in $O(n + |\mathsf{MAW}(T)|)$ time and $O(n)$ working space. Fujishige et al.'s algorithm [44] can easily be modified so it uses $O(|\mathsf{MAW}(T)|)$ time for reporting all MAWs, by explicitly storing all MAWs when $|\mathsf{MAW}(T)| \in O(n)$. The key tool used in these two algorithms is an $O(n)$-size automaton called the ***DAWG*** [15], which accepts all substrings of T. The DAWG for string T can be built in $O(n \log \sigma)$ time for general ordered alphabets [15], or in $O(n)$ time for integer alphabets of size polynomial in n [44]. Charalampopoulos et al. [20] presented an algorithm that computes all MAWs in output-sensitive time without using the DAWG of T. Belazzougui et al. [11] showed that $\mathsf{MAW}(T)$ can also be computed in $O(n + |\mathsf{MAW}(T)|)$ time, provided that the *bidirectional Burrows-Wheeler transform* of a given string has already been computed. Barton et al. [8] proposed a practical algorithm to compute $\mathsf{MAW}(T)$ in $\Theta(n\sigma)$ time and working space[3] based on the *suffix array* [75] of T. A parallel algorithm for computing MAWs has also been proposed [9]. Fici and Gawrychowski [42] extended the notion of MAWs to rooted/unrooted labeled trees and presented efficient algorithms to compute them.

[3]The original claimed bound in [8] is $O(n)$, however, the authors assumed that $\sigma \in O(1)$.

1.6 Paper organization

Chapter 2 introduces necessary notations. In Chapter 3, we first present matching upper bounds and lower bounds on the number of MAWs to be added/deleted when the current window is shifted to the right by one character. Next, we show matching upper bounds and lower bounds on the number of MAWs in the RLE strings. In the last section of Chapter 3, we present matching upper bounds and lower bounds of the number of MAWs to be added/deleted in the case of sliding window and RLE strings. In Chapter 4, we show how to compute our new grammar index and show how one can locate all pattern occurrences efficiently. Implementation and experiments are summarized in the last section of this chapter. In Chapter 5, we present the worst-case sensitivity of string compressors and repetitiveness measures: Section 5.1 deals with the GCIS grammar g_{is}; Section 5.3 deals with the LZ-End z_{End}; Section 5.4 deals with the smallest grammar g^*, and its applications to approximation grammars such as AVL-grammars, α-balanced grammars, and Recompression; Section 5.2 deals with the Bisection grammar g_{bsc}; In Chapter 6 we conclude the thesis and list several open questions of interest.

Chapter 2

Preliminaries

2.1 Strings

Let Σ be an integer *alphabet* of size σ. An element of Σ is called a *character*. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. If $T = xyz$, then x, y, and z are called a *prefix*, *substring*, and *suffix* of T, respectively. They are called a *proper prefix*, *proper substring*, and *proper suffix* of T if $x \neq T$, $y \neq T$, and $z \neq T$, respectively.

For any $1 \leq i \leq |T|$, the i-th character of T is denoted by $T[i]$. For any $1 \leq i \leq j \leq |T|$, $T[i..j]$ denotes the substring of T starting at i and ending at j. For convenience, let $T[i..j] = \varepsilon$ for $0 \leq j < i \leq |T| + 1$. For any $i \leq |T|$ and $1 \leq j$, let $T[..i] = T[1..i]$ and $T[j..] = T[j..|T|]$.

We say that a string w *occurs* in a string T if w is a substring of T. Note that by definition the empty string ε is a substring of any string T and hence ε always occurs in T.

Let $\#_T w$ denote the number of occurrences of a string w in a string T. We will abbreviate it to $\#w$ when no confusion occurs.

2.2 Run Length Encoding (RLE)

The *run-length encoding* $\mathsf{rle}(T)$ of string T is a compact representation of T such that each maximal run of the same characters in T is represented by a pair of the character and the length of the maximal run. More formally, $\mathsf{rle}(T) = a_1^{p_1} \cdots a_m^{p_m}$ encodes each substring $T[i..i+p-1]$ by a^p if $T[j] = a \in \Sigma$ for every $i \leq j \leq i+p-1$, $T[i-1] \neq T[i]$, and $T[i+p-1] \neq T[i+p]$. Each a^p in $\mathsf{rle}(T)$ is called a (character) *run*, and p is called the exponent of this run. The j-th maximal run in $\mathsf{rle}(T)$ is denoted by r_j, namely $\mathsf{rle}(T) = r_1 \cdots r_m$. The *size* of $\mathsf{rle}(T)$, denoted $R(T)$, is the number of maximal character runs in $\mathsf{rle}(T)$. E.g., for a string $T = \mathtt{aacccccccbbabbbb}$ of length 18, $\mathsf{rle}(T) = \mathtt{a}^2\mathtt{c}^7\mathtt{b}^2\mathtt{a}^1\mathtt{b}^4$ and $R(T) = 5$.

Our model of computation is a standard word RAM with machine word size $\Omega(\log |T|)$, and the space requirements of our data structures will be measured by the number of words (not bits). Thus, $\mathsf{rle}(T)$ of size m can be stored in $O(m)$ space.

2.2.1 Bridges

A string $w \in \Sigma^*$ of length $|w| \geq 2$ is said to be a *bridge* if $w[1] \neq w[2]$ and $w[|w|-1] \neq w[|w|]$. In other words, both of the first run and the last run in $\mathsf{rle}(w)$ are of length 1. A substring of T that is a bridge is called a bridge substring of T. Let B_ℓ denote the set of bridge substrings w of T with $R(w) = \ell$. Further let $\mathcal{B} = \bigcup_\ell B_\ell$ be the set of all bridge substrings of T. For example, for the same string $T = \mathtt{aacccccccbbabbbb}$ as the above one, the substring $\mathtt{a}\mathtt{c}^7\mathtt{b}^2\mathtt{a}$ of T is a bridge, and $B_4 = \{\mathtt{a}\mathtt{c}^7\mathtt{b}^2\mathtt{a}, \mathtt{c}\mathtt{b}^2\mathtt{a}^1\mathtt{b}\}$. For a string w with $R(w) \geq 3$, we can obtain a bridge substring of w by removing the first and the last runs of w and then *shrinking* the runs at both ends so that their exponents are 1. We denote by $\mathsf{shk}(w)$ such shrunk bridge. For convenience, let $\mathsf{shk}(w) = \varepsilon$ if $R(w) \leq 2$. Also, for every $k \geq 2$, we denote $\mathsf{shk}^k(w) = \mathsf{shk}(\mathsf{shk}^{k-1}(w))$. For example, consider the same T as the above again, $\mathsf{shk}(T) = \mathtt{cbba}$, $\mathsf{shk}^2(w) = \mathtt{b}$, and $\mathsf{shk}^k(w) = \varepsilon$ for any $k \geq 3$.

In the rest of this thesis, we will consider an arbitrarily fixed string T of length n. For convenience, we assume that $n \geq 3$ and that there are special terminal symbols $T[1] = T[n] = \$ \notin \Sigma$ not occurring inside T. Since $\$ \notin \Sigma$, we do not consider any MAW containing \$ for T in our arguments to follow (recall that a MAW must be an element of Σ^*). In addition, since \$ does not occur elsewhere in T, $\mathsf{MAW}(T) = \mathsf{MAW}(T[2..n-1])$ holds.

Example 2.1. *Consider* $T = \$\mathtt{b}^2\mathtt{a}\mathtt{c}^3\mathtt{b}\mathtt{a}^2\$ = \$\mathtt{bbacccbaa}\$$. *All MAWs in* $\mathsf{MAW}(T)$ *are divided into the following five types:* $\mathcal{M}_1 = \{\mathtt{aaa}, \mathtt{bbb}, \mathtt{cccc}\}$*;* $\mathcal{M}_2 = \{\mathtt{ca}, \mathtt{bc}\}$*;* $\mathcal{M}_3 = \{\mathtt{acb}, \mathtt{accb}\}$*;* $\mathcal{M}_4 = \{\mathtt{cbac}\}$*;* $\mathcal{M}_5 = \{\mathtt{bbaa}\}$.

Let Σ' denote the set of characters occurring in T except for \$. Let $\sigma' = |\Sigma'|$ be the number of distinct characters occurring in $T[2..n-1]$.

2.2.2 Truncated RLE Suffix Array (tRLESA)

We explain suffix array(SA) and *truncated RLE Suffix Array* (*tRLESA*)[104]. A suffix s of T is called a truncated RLE(tRLE) suffix of T if $s = \mathsf{tRLE}(i) = a_i r_{i+1} \cdots r_m$ where the first

a_i is the last character in the previous run r_i. SA for T is an integer array of length n such that $\mathsf{SA}(T)[i] = k$ iff $T[k..]$ is the i-th lexicographically smallest suffix for T. $\mathsf{tRLESA}(T)$ for $\mathsf{rle}(T) = r_1 \cdots r_m$ is an integer array of length m such that $\mathsf{tRLESA}(T)[i] = k$ iff $a_k r_{k+1} \cdots r_m$ is the i-th lexicographically smallest tRLE suffix for T. tRLESA occupies $O(m)$ space, and can be built in $O(m \log m)$ time with $O(m)$ working space [104]. Let $\mathsf{LCP}(x, y)$ be the longest common prefix of x and y. $\mathsf{tRLELCP}(T)$ for $\mathsf{rle}(T) = r_1 \cdots r_m$ is an integer array of length m such that $\mathsf{tRLELCP}(T)[i] = k$ iff $R(\mathsf{LCP}(\mathsf{tRLE}(\mathsf{tRLESA}[i]), \mathsf{tRLE}(\mathsf{tRLESA}[i+1]))) = k$. $\mathsf{tRLELCP}$ occupies $O(m)$ space and can calculate in $O(m \log m)$ time [104].

Example 2.2. *Let* $T = \texttt{abbabbaa}\#$. *Then,*

$$\begin{aligned}
T[1..] &= \texttt{abbabbaa}\#,\\
T[2..] &= \texttt{bbabbaa}\#,\\
T[3..] &= \texttt{babbaa}\#,\\
T[4..] &= \texttt{abbaa}\#,\\
T[5..] &= \texttt{bbaa}\#,\\
T[6..] &= \texttt{baa}\#,\\
T[7..] &= \texttt{aa}\#,\\
T[8..] &= \texttt{a}\#,\\
T[9..] &= \#,
\end{aligned}$$

The following order is obtained by sorting the suffixes in lexicographical order.

$$\begin{aligned}
T[9..] &= \#,\\
T[8..] &= \texttt{a}\#,\\
T[7..] &= \texttt{aa}\#,\\
T[4..] &= \texttt{abbaa}\#,\\
T[1..] &= \texttt{abbabbaa}\#,\\
T[6..] &= \texttt{baa}\#,\\
T[3..] &= \texttt{babbaa}\#,\\
T[5..] &= \texttt{bbaa}\#,\\
T[2..] &= \texttt{bbabbaa}\#,
\end{aligned}$$

Therefore, the suffix array is $SA = [10, 9, 8, 3, 7, 2, 1, 6, 5, 4]$. *In addition, we also gain the* *following order by sorting the truncated suffixes in lexicographical order:*

$$\begin{aligned}
T[9..] = \mathsf{tRLE}(6) &= \#, \\
T[8..] = \mathsf{tRLE}(5) &= \mathtt{a}\#, \\
T[4..] = \mathsf{tRLE}(3) &= \mathtt{abbaa}\#, \\
T[1..] = \mathsf{tRLE}(1) &= \mathtt{abbabbaa}\#, \\
T[6..] = \mathsf{tRLE}(4) &= \mathtt{baa}\#, \\
T[3..] = \mathsf{tRLE}(2) &= \mathtt{babbaa}\#,
\end{aligned}$$

Therefore, the truncated suffix array is $tRLESA = [6, 5, 3, 1, 4, 2]$. *In this example,*

$$tRLELCP[3] = R(\mathsf{LCP}(T[4..], T[1..])) = R(\mathtt{abba}) = 3.$$

2.3 Minimal Absent Words (MAWs)

A string $w \in \Sigma^*$ is called an *absent word* for a string T if w does not occur in T, namely if $\#w = 0$. An absent word w for T is called a *minimal absent word* or *MAW* for T if all proper substrings of w occur in T.

We note that if w is a string of length 1 which does not occur in T (i.e. w is a single character in Σ of size σ not occurring in T), then w is a MAW for T since $w[2..] = w[..|w| - 1] = \varepsilon$ is a substring of T.

We denote by $\mathsf{MAW}(T)$ the set of all MAWs for T. By the definition of MAWs, it is clear that $w = aub \in \mathsf{MAW}(T)$ iff the three following conditions hold:

(A) $\#(aub) = 0$.

(B) $\#(ub) \geq 1$.

(C) $\#(au) \geq 1$.

For a MAW of length 1 (namely a character not occurring in T), we use a convention that $u = \varepsilon$ and a and b are united into a single character.

Example 2.3. *Let* $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. *Then, the set of MAWs for string* $\mathtt{cbaaaa}$ *is:*

$$\mathsf{MAW}(\mathtt{cbaaaa}) = \{\mathtt{cc}, \mathtt{bb}, \mathtt{aaaaa}, \mathtt{bc}, \mathtt{ab}, \mathtt{ca}, \mathtt{ac}, \mathtt{d}\}.$$

An alternative definition of MAWs is such that a string aub of length at least two with $a, b \in \Sigma$ and $u \in \Sigma^*$ is a MAW of T if $\#(aub) = 0$, $\#(au) \geq 1$ and $\#(ub) \geq 1$. For a MAW of length 1 (namely a character not occurring in T), we use a convention that $u = \varepsilon$ and a and b are united into a single character.

The MAWs in $\mathsf{MAW}(T)$ are partitioned into the following five disjoint subsets $\mathcal{M}_i$ ($1 \leq i \leq 5$) based on their RLE sizes $R(aub)$:

- $\mathcal{M}_1 = \{aub \in \mathsf{MAW}(T) \mid R(aub) = 1\}$;
- $\mathcal{M}_2 = \{aub \in \mathsf{MAW}(T) \mid R(aub) = 2, u = \varepsilon\}$;
- $\mathcal{M}_3 = \{aub \in \mathsf{MAW}(T) \mid R(aub) = 3, a \neq u[1]$ and $b \neq u[|u|]\}$;
- $\mathcal{M}_4 = \{aub \in \mathsf{MAW}(T) \mid R(aub) \geq 4, a \neq u[1]$ and $b \neq u[|u|]\}$;
- $\mathcal{M}_5 = \{aub \in \mathsf{MAW}(T) \mid R(aub) \geq 2, a = u[1]$ or $b = u[|u|]\}$.

For $1 \leq i \leq 5$, a MAW aub in $\mathcal{M}_i$ is called of *Type-i*.

2.3.1 MAWs for sliding window

Given a string T of length n and a sliding window $S_i = T[i..j]$ of length $d = j - i + 1$ for increasing $i = 1, \ldots, n-d+1$, our goal is to analyze how many MAWs for the sliding window can change when the window shifts over the string T. We will consider both the maximum change per one shift, and the maximum total number of changes when sliding the window from the beginning to the end.

As was done in [30], for simplicity, we separately consider two symmetric operations of appending a new character to the right of the window and of deleting the leftmost character from the window.

Example 2.4. *Let* $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. *Consider appending character* $\mathtt{c}$ *to the right of string* $\mathtt{cbaaaa}$. *Then,*

$$\begin{aligned}
\mathsf{MAW}(\mathtt{cbaaaa}) &= \{\mathtt{cc}, \mathtt{bb}, \mathtt{aaaaa}, \mathtt{bc}, \mathtt{ab}, \mathtt{ca}, \mathtt{d}, \mathtt{ac}\}, \\
\mathsf{MAW}(\mathtt{cbaaaac}) &= \{\mathtt{cc}, \mathtt{bb}, \mathtt{aaaaa}, \mathtt{bc}, \mathtt{ab}, \mathtt{ca}, \mathtt{dacb}, \mathtt{bac}, \mathtt{baac}, \mathtt{baaac}\}.
\end{aligned}$$

Thus $\mathsf{MAW}(\mathtt{cbaaaa}) \triangle \mathsf{MAW}(\mathtt{cbaaaac}) = \{\underline{\mathtt{ac}}, \mathtt{acb}, \mathtt{bac}, \mathtt{baac}, \mathtt{baaac}\}$*, where the underlined string is deleted from and the strings without underlines are added to the set of MAWs by appending* $\mathtt{c}$ *to* $\mathtt{cbaaaa}$.

2.4 Factorizations and grammars

For any non-negative integer n, let Σ^n denote the set of strings of length n over Σ.

A factorization of a non-empty string T is a sequence $f_1, \dots, f_x$ of non-empty substrings of T such that $T = f_1 \cdots f_x$. Each f_i is called a *factor*. The *size* of the factorization is the number x of factors in the factorization.

A context-free grammar $\mathcal{G}$ which generates only a single string T is called a *grammar compression* for T. The *size* of $\mathcal{G}$ is the total length of the right-hand sides of all the production rules in $\mathcal{G}$. The *height* of $\mathcal{G}$ is the height of the derivation tree of $\mathcal{G}$.

2.5 Suffix type

SAIS [92] is a linear-time algorithm for computing the suffix array [75]. We briefly review the parts of SAIS important for constructing the GCIS grammar. SAIS assigns each suffix a type, which is either $\texttt{L}$ or $\texttt{S}$: $T[i..]$ is an $\texttt{L}$ suffix if $T[i..] \succ T[i+1..]$, or $T[i..]$ is an $\texttt{S}$ suffix otherwise, i.e., $T[i..] \prec T[i+1..]$, where we stipulate that $T[|T|]$ is always type $\texttt{S}$. Since it is not possible that $T[i..] = T[i+1..]$, SAIS assigns each suffix a type. An $\texttt{S}$ suffix $T[i..]$ is additionally an $\texttt{S}^*$ suffix (also called LMS suffix in [92]) if $T[i-1..]$ is an $\texttt{L}$ suffix. The substring between two succeeding $\texttt{S}^*$ suffixes is called an *LMS substring*. In other words, a substring $T[i..j]$ with $i < j$ is an LMS substring if and only if $T[i..]$ and $T[j..]$ are $\texttt{S}^*$ suffixes and there is no $k \in (i..j)$ such that $T[k..]$ is an $\texttt{S}^*$ suffix. A border case is $T[|T|..]$, which has to be the smallest suffix of T (and can be achieved by appending an artificial character $\texttt{\$}$ to T lexicographically smaller than all other characters appearing it T) such that $T||T|..]$ in an $\texttt{S}^*$ suffix. We additionally treat $T[|T|..]$ as an LMS substring. Regarding the defined types, we make no distinction between suffixes and their starting positions (e.g., the statements that (a) $T[i]$ is type $\texttt{L}$ and (b) $T[i..]$ is an $\texttt{L}$ suffix are equivalent). In fact, we can determine $\texttt{L}$ and $\texttt{S}$ positions solely based on their succeeding positions with the equivalent definition: $T[i]$ is $\texttt{L}$ if $T[i] > T[i+1]$, $T[i]$ is $\texttt{S}$ if $T[i] < T[i+1]$, or $T[i]$ has the same type as $T[i+1]$ if $T[i] = T[i+1]$.

Nong et al. [92, A3.4] compute the lexicographic order of all LMS substrings with the induced sorting, which we omit here since we use it as a black box.

2.6 Grammar Compression by Induced Sorting (GCIS)

With $\lg$ we denote the logarithm $\log_2$ to base two. We define that the element of Σ is small enough to fit into a constant number of machine words.

The LMS substrings of $\#T$ for $\#$ being a special character smaller than all characters appearing in T, induce a factorization of $T = F_1 \cdots F_z$, where each factor starts with an LMS substring. We call this factorization *LMS-factorization*. By replacing each factor F_i by the lexicographic rank of its respective LMS substring, we obtain a string $T^{(1)}$ of these ranks. We recurse on $T^{(1)}$ until we obtain a string $T^{(\tau_T - 1)}$ whose characters are all unique or whose LMS-factorization consists of at most two factors.

Example 2.5. *Let* $\Sigma = \{\mathtt{1}, \mathtt{2}, \mathtt{3}\}, T = \mathtt{1231312313}\#$*. Then, GCIS factorizes it like this:*

$$\mathtt{123}|\mathtt{13}|\mathtt{123}|\mathtt{13}|\#$$

and then GCIS make rules:

$$\begin{aligned} \mathtt{4} &\rightarrow \# \\ \mathtt{5} &\rightarrow \mathtt{123} \\ \mathtt{6} &\rightarrow \mathtt{13} \end{aligned}$$

and we obtain $T^{(1)} = \mathtt{56564}$. We recurse the same algorithm on it, then we GCIS makes the rule below:

$$\begin{aligned} \mathtt{7} &\rightarrow \mathtt{4} \\ \mathtt{8} &\rightarrow \mathtt{56} \end{aligned}$$

and we obtain $T^{(2)} = \mathtt{887}$. There are only 2 factors in next GCIS factorization $(88, 7)$, so GCIS terminates the recursion.

2.7 Worst-case sensitivity of compressors and repetitiveness measures

For a string compression algorithm C and an input string T, let $C(T)$ denote the size of the compressed representation of T obtained by applying C to T. For convenience, we use the same notation when C is a string repetitiveness measure, namely, $C(T)$ is the value of the measure C for T.

Let us consider the following edit operations on strings: character substitution (sub), character insertion (ins), and character deletion (del). For two strings T and S, let $\mathsf{ed}(T, S)$ denote the *edit distance* between T and S, namely, $\mathsf{ed}(T, S)$ is the minimum number of edit operations that transform T into S.

Our interest in this thesis is: "How much can the compression size or the repetitiveness measure size change when a single-character-wise edit operation is performed on a string?" To answer this question, for a given string length n, we consider an arbitrarily fixed string T of length n and all strings T' that can be obtained by applying a single edit operation to T, that is, $\mathsf{ed}(T, T') = 1$. We define the worst-case *multiplicative sensitivity* of C w.r.t. a substitution, insertion, and deletion as follows:

$$\begin{aligned}
\mathsf{MS}_{\mathrm{sub}}(C, n) &= \max_{T \in \Sigma^n} \{C(T')/C(T) : T' \in \Sigma^n, \mathsf{ed}(T, T') = 1\},\\
\mathsf{MS}_{\mathrm{ins}}(C, n) &= \max_{T \in \Sigma^n} \{C(T')/C(T) : T' \in \Sigma^{n+1}, \mathsf{ed}(T, T') = 1\},\\
\mathsf{MS}_{\mathrm{del}}(C, n) &= \max_{T \in \Sigma^n} \{C(T')/C(T) : T' \in \Sigma^{n-1}, \mathsf{ed}(T, T') = 1\}.
\end{aligned}$$

We also consider the worst-case *additive sensitivity* of C w.r.t. a substitution, insertion, and deletion, as follows:

$$\begin{aligned}
\mathsf{AS}_{\mathrm{sub}}(C, n) &= \max_{T \in \Sigma^n} \{C(T') - C(T) : T' \in \Sigma^n, \mathsf{ed}(T, T') = 1\},\\
\mathsf{AS}_{\mathrm{ins}}(C, n) &= \max_{T \in \Sigma^n} \{C(T') - C(T) : T' \in \Sigma^{n+1}, \mathsf{ed}(T, T') = 1\},\\
\mathsf{AS}_{\mathrm{del}}(C, n) &= \max_{T \in \Sigma^n} \{C(T') - C(T) : T' \in \Sigma^{n-1}, \mathsf{ed}(T, T') = 1\}.
\end{aligned}$$

We remark that, in general, $C(T')$ can be larger than $C(T)$ even when T' is obtained by a character deletion from T (i.e. $|T'| = n - 1$). Such strings T are already known for the Lempel-Ziv 77 factorization size z when $T' = T[2..n]$ [80], or for the smallest string attractor size γ when $T' = T[1..n-1]$ [76].

The above remark implies that in general the multiplicative/additive sensitivity for insertions and deletions may not be symmetric and therefore they need to be discussed separately for some C. Note, on the other hand, that the maximum difference between $C(T')$ and $C(T)$ when $|T'| = n - 1$ (deletion) and $C(T') - C(T) < 0$ is equivalent to $\mathsf{AS}_{\mathrm{ins}}(C, n-1)$, and symmetrically the maximum difference of $C(T')$ and $C(T)$ when $|T'| = n + 1$ (insertion) and $C(T') - C(T) < 0$ is equivalent to $\mathsf{AS}_{\mathrm{del}}(C, n+1)$, with the roles of T and T' exchanged. Similar arguments hold for the multiplicative sensitivity with insertions/deletions. Consequently, it suffices to consider $\mathsf{MS}_{\mathrm{ins}}(C, n)$, $\mathsf{MS}_{\mathrm{del}}(C, n)$, $\mathsf{AS}_{\mathrm{ins}}(C, n)$, $\mathsf{AS}_{\mathrm{del}}(C, n)$ for insertions/deletions.

Consider two measures α and β. An upper bound for the multiplicative sensitivity of β can readily be derived in the some cases, as follows:

Lemma 2.1. *Let T be any string of length n and let T' be any string with $\mathsf{ed}(T, T') = 1$. If the following conditions:*

- $\alpha(T')/\alpha(T) \in O(1)$*;*
- $\alpha(T) \leq \beta(T)$*;*
- $\beta(T) \in O(\alpha(T) \cdot f \cdot (n, \alpha(T)))$*, where f is a function such that for any constant c there exists a constant c' satisfying $f(n, c \cdot \alpha(T)) \leq c' \cdot f(n, \alpha(T))$.*

all hold, then we have the following upper bounds (1), (2), and (3) for the sensitivity of β:

(1) $\mathsf{MS}_{\mathrm{sub}}(\beta, n) \in O(f(n, \alpha))$ *and* $\mathsf{AS}_{\mathrm{sub}}(\beta, n) \in O(\alpha \cdot f(n, \alpha))$*;*

(2) $\mathsf{MS}_{\mathrm{ins}}(\beta, n) \in O(f(n, \alpha))$ *and* $\mathsf{AS}_{\mathrm{ins}}(\beta, n) \in O(\alpha \cdot f(n, \alpha))$*;*

(3) $\mathsf{MS}_{\mathrm{del}}(\beta, n) \in O(f(n, \alpha))$ *and* $\mathsf{AS}_{\mathrm{del}}(\beta, n) \in O(\alpha \cdot f(n, \alpha))$*.*

Proof. Let $c = \alpha(T')/\alpha(T)$, where c is a constant. Then we have

$$\begin{array}{rcl}
\frac{\beta(T')}{\beta(T)} & \in & O\left(\frac{\alpha(T') \cdot f(n, \alpha(T'))}{\alpha(T)}\right) \\
& \subseteq & O\left(\frac{\alpha(T') \cdot f(n, c \cdot \alpha(T))}{\alpha(T)}\right) \\
& \subseteq & O\left(\frac{\alpha(T') \cdot c' \cdot f(n, \alpha(T))}{\alpha(T)}\right) \\
& \subseteq & O(f(n, \alpha(T))).
\end{array}$$

Also,

$$\begin{array}{rcl}
\beta(T') - \beta(T) & \in & O(\alpha(T') \cdot f(n, \alpha(T')) - \alpha(T) \cdot f(n, \alpha(T))) \\
& \subseteq & O(\alpha(T') \cdot f(n, c \cdot \alpha(T)) - \alpha(T) \cdot f(n, \alpha(T))) \\
& \subseteq & O(\alpha(T') \cdot c' \cdot f(n, \alpha(T)) - \alpha(T) \cdot f(n, \alpha(T))) \\
& \subseteq & O((c' \cdot \alpha(T') - \alpha(T)) \cdot f(n, \alpha(T))) \\
& \subseteq & O((c' \cdot c \cdot \alpha(T) - \alpha(T)) \cdot f(n, \alpha(T))) \\
& \subseteq & O(\alpha(T) \cdot f(n, \alpha(T))).
\end{array}$$

□

The functions satisfying $f(n, c \cdot \alpha(T)) \leq c' \cdot f(n, \alpha(T))$ include functions f which are polynomial, poly-logarithmic, or constant in terms of $\alpha(T)$.

Chapter 3

Combinatorics and Efficient Algorithms for MAWs

3.1 MAWs for sliding window

In this section, we present our new and tight bounds for the changes of MAWs for the sliding window over the string T. In Section 3.1.1 we consider the case of general alphabets of size σ. In Section 3.1.2 we obtain tighter upper bounds in the case of binary alphabets where $\sigma = 2$.

3.1.1 Tight bounds on the changes to MAWs for sliding window

First, we look at the case of non-binary strings. we consider the number of changes of MAWs when the current window $T[i..j]$ is extended by adding a new character $T[j + 1]$. After that, we show that leftmost removal is for the symmetric case where the leftmost character $T[i]$ is deleted from $T[i..i + j + 1]$. Finally, we consider the total number of changes of MAWs while the window has been shifted from the beginning of T until its end.

Changes to MAWs when a character is appended to the right

We consider the number of changes of MAWs when appending $T[j + 1]$ to the current window $T[i..j]$.

For the number of deleted MAWs, the next lemma is known:

Lemma 3.1 ([30])**.** *For any* $1 \leq i \leq j < n$, $|\mathsf{MAW}(T[i..j]) \setminus \mathsf{MAW}(T[i..j + 1])| = 1$.

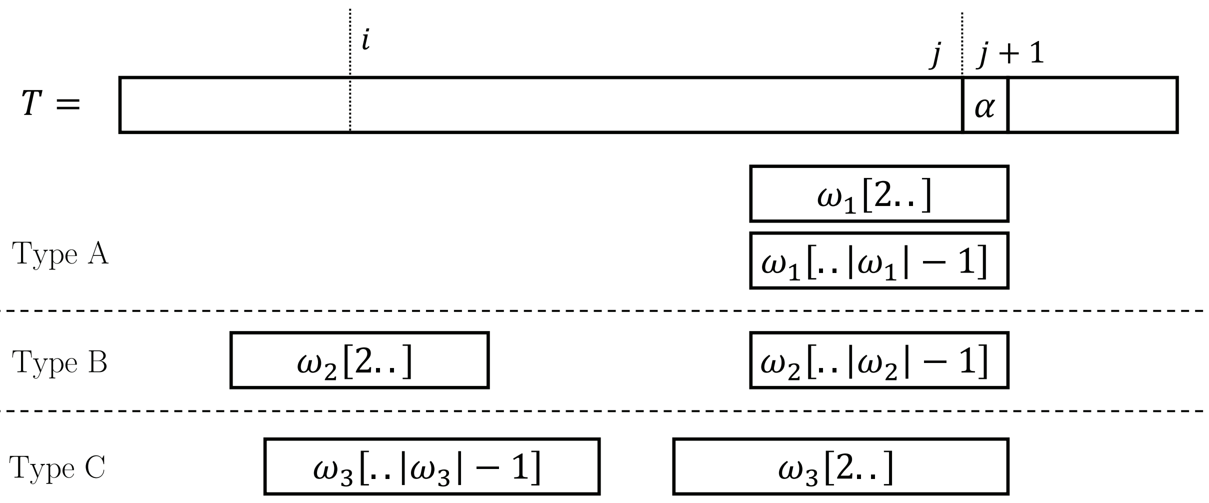


Figure 3.1: Illustration for the three types of MAWs, where $w_1 \in \mathcal{M}_A$, $w_2 \in \mathcal{M}_B$, and $w_3 \in \mathcal{M}_C$.

Next, we consider the number of added MAWs. We classify the MAWs in $\mathsf{MAW}(T[i..j+1]) \setminus \mathsf{MAW}(T[i..j])$ to the following three types[1] (see Figure 3.1). A MAW w in $\mathsf{MAW}(T[i..j+1]) \setminus \mathsf{MAW}(T[i..j])$ is said to be of:

Type-A if neither $w[2..]$ nor $w[..|w|-1]$ occurs in $T[i..j]$;

Type-B if $w[2..]$ occurs in $T[i..j]$ but $w[..|w|-1]$ does not occur in $T[i..j]$;

Type-C if $w[2..]$ does not occur in $T[i..j]$ but $w[..|w|-1]$ occurs in $T[i..j]$.

We denote by $\mathcal{M}_A$, $\mathcal{M}_B$, and $\mathcal{M}_C$ the sets of MAWs of Type-A, Type-B and Type-C, respectively. Recall that w is a MAW for $T[i..j+1]$.

Let $\sigma_{i,j}$ be the number of distinct characters occurring in the current window $T[i..j]$.

The next three lemmas show the upper bounds of $\mathcal{M}_A$, $\mathcal{M}_B$, and $\mathcal{M}_C$:

Lemma 3.2 ([30]). *For any $1 \le i \le j < n$, $|\mathcal{M}_A| \le 1$. Also, if α is the character appended to $T[i..j]$, then the only element of $\mathcal{M}_A$ is of the form α^k for some $k \ge 1$.*

Lemma 3.3. *For any $1 \le i \le j < n$, $|\mathcal{M}_B| \le \sigma_{i,j}$.*

Proof. It is shown in [30] that the last characters of all MAWs in $\mathcal{M}_B$ are all distinct. Furthermore, by the definition of $\mathcal{M}_B$, the last character $T[j+1]$ of each MAW in $\mathcal{M}_B$ must occur in the current window $T[i..j]$. Thus, $|\mathcal{M}_B| \le \sigma_{i,j}$. □

[1]At least one of $w[2..]$ and $w[..|w|-1]$ does not occur in $T[i..j]$, since $w \notin \mathsf{MAW}(T[i..j])$.

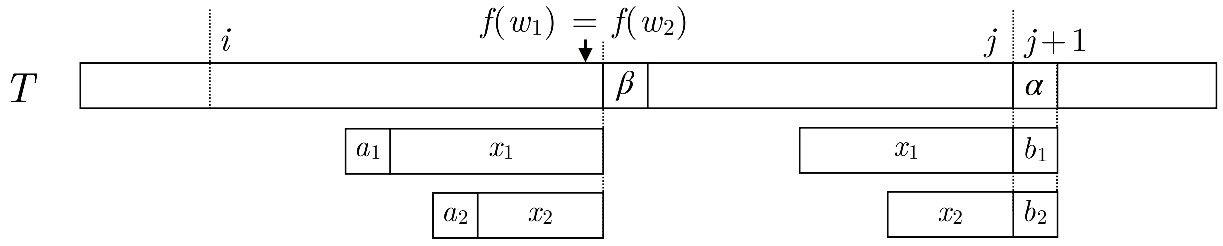


Figure 3.2: Illustration for the contradiction in the proof of Lemma 3.5. Consider two strings $w_1 = a_1x_1b_1$ and $w_2 = a_2x_2b_2$ that are MAWs for T of Type-C where $a_1, a_2, b_1, b_2 \in \Sigma$ and $x_1, x_2 \in \Sigma^*$. If $|w_1| > |w_2|$ and $f(w_1) = f(w_2)$, then x_2 is a proper suffix of x_1, and it contradicts that $a_2x_2b_2$ is absent from T.

Lemma 3.4. *For any* $1 \le i \le j < n$, $|\mathcal{M}_C| \le d - 1$, *where* $d = j - i + 1$.

Proof. We show that there is an injection $f : \mathcal{M}_C \to [i, j-1]$ which maps each MAW $w \in \mathcal{M}_C$ to the ending position of the leftmost occurrence of $w[..|w|-1]$ in the current window $T[i..j]$.

First, we show that the range of this function f is $[i, j-1]$. By definition, w is absent from $T[i..j+1]$ and $w[|w|] = T[j+1]$ for each $w \in \mathcal{M}_C$, and thus, no occurrence of $w[..|w|-1]$ in $T[i..j]$ ends at position j. Hence, the range of f does not contain the position j, i.e. it is $[i, j-1]$.

Next, for the sake of contradiction, we assume that f is not an injection, i.e. there are two distinct MAWs $w_1, w_2 \in \mathcal{M}_C$ such that $f(w_1) = f(w_2)$. Without loss of generality, assume $|w_1| \ge |w_2|$. Since $w_1[|w_1|] = w_2[|w_2|] = T[j+1]$ and $f(w_1) = f(w_2)$, w_2 is a suffix of w_1. If $|w_1| = |w_2|$, then $w_1 = w_2$ and it contradicts with $w_1 \ne w_2$. If $|w_1| > |w_2|$, then w_2 is a proper suffix of w_1, and it contradicts with the fact that w_2 is absent from $T[i..j+1]$ (see Figure 3.2). Therefore, f is an injection and $|\mathcal{M}_C| \le j - 1 - i + 1 = d - 1$. □

Summing up all the upper bounds for $\mathcal{M}_A$, $\mathcal{M}_B$, and $\mathcal{M}_C$, we obtain the following lemma:

Lemma 3.5. *For any* $1 \le i \le j < n$, $|\mathsf{MAW}(T[i..j+1]) \setminus \mathsf{MAW}(T[i..j])| \le \sigma_{i,j} + d$, *where* $d = j - i + 1$.

Proof. Immediately follows from Lemmas 3.2, 3.3, and 3.4 and that $\mathcal{M}_A$, $\mathcal{M}_B$, and $\mathcal{M}_C$ are mutually disjoint. □

Now we obtain the main result of this subsection, which shows the matching upper and lower bounds for $|\mathsf{MAW}(T[i..j+1]) \triangle \mathsf{MAW}(T[i..j])|$.

Theorem 3.1. *For any* $1 \le i \le j < n$, $|\mathsf{MAW}(T[i..j+1]) \triangle \mathsf{MAW}(T[i..j])| \le \sigma_{i,j} + d + 1$, *where* $d = j - i + 1$. *The upper bound is tight when* $\sigma \ge 3$ *and* $\sigma_{i,j} + 1 \le \sigma$.

Proof. By Lemma 3.1 and Lemma 3.5, we have $|\mathsf{MAW}(T[i..j+1]) \triangle \mathsf{MAW}(T[i..j])| = |\mathsf{MAW}(T[i..j+1]) \setminus \mathsf{MAW}(T[i..j])| + |\mathsf{MAW}(T[i..j]) \setminus \mathsf{MAW}(T[i..j+1])| \leq \sigma_{i,j} + d + 1$.

In the following, we show that the upper bound is tight, i.e. there is a string Z of length d and a character α, where $|\mathsf{MAW}(Z) \triangle \mathsf{MAW}(Z\alpha)| = \sigma_{1,d} + d + 1$ for any two integers d and $\sigma_{1,d}$ with $1 \leq \sigma_{1,d} \leq d$ and $\sigma_{1,d} + 1 \leq \sigma$. Namely, in this example, we set $i = 1$ and $j = d$. Let $\Sigma = \{a_1, a_2, \cdots, a_\sigma\}$ be an alphabet. Given two integers d and $\sigma_{1,d}$ with $1 \leq \sigma_{1,d} \leq d$ and $\sigma_{1,d} + 1 \leq \sigma$, consider a string $Z = a_1 a_2 \cdots a_{\sigma_{1,d}-1} a_{\sigma_{1,d}}^{d-\sigma_{1,d}+1}$ of length d and a character $\alpha = a_{\sigma_{1,d}+1}$. Then,

$$\mathsf{MAW}(Z) \setminus \mathsf{MAW}(Z\alpha) = \{\alpha\}.$$

Also,

$$\begin{aligned} \mathsf{MAW}(Z\alpha) \setminus \mathsf{MAW}(Z) &= \{\alpha^2\} \cup \{\alpha a_i \mid 1 \leq i \leq \sigma_{1,d}\} \cup \{a_i \alpha \mid 1 \leq i \leq \sigma_{1,d} - 1\} \\ &\quad \cup \{a_{\sigma_{1,d}-1} a_{\sigma_{1,d}}^{e} \alpha \mid 1 \leq e \leq d - \sigma_{1,d}\}. \end{aligned}$$

This leads to the matching lower bound $|\mathsf{MAW}(Z) \triangle \mathsf{MAW}(Z\alpha)| = \sigma_{1,d} + d + 1$. □

A concrete example for our lower-bound strings Z and $Z\alpha$ is shown below.

Example 3.1. *Consider a string $Z =$ `abcddd` with $\sigma_{1,6} = 4$, and let $d = |Z| = 6$. We have $d - \sigma_{1,6} + 1 = 3$. Also, let $\alpha =$ `e`. Then,*

$$\mathsf{MAW}(\texttt{abcddd}) \setminus \mathsf{MAW}(\texttt{abcddde}) = \{\texttt{e}\}$$

and

$$\begin{aligned} &\mathsf{MAW}(\texttt{abcddde}) \setminus \mathsf{MAW}(\texttt{abcddd}) \\ &= \mathcal{M}_A \cup \mathcal{M}_B \cup \mathcal{M}_C \\ &= \{\texttt{ee}\} \cup \{\texttt{ea}, \texttt{eb}, \texttt{ec}, \texttt{ed}\} \cup \{\texttt{ae}, \texttt{be}, \texttt{ce}, \texttt{cde}, \texttt{cdde}\}, \end{aligned}$$

and therefore $|\mathsf{MAW}(Z) \triangle \mathsf{MAW}(Z\alpha)| = \sigma_{1,6} + d + 1 = 11$.

Changes to MAWs when a character already occurring in the window is added to the right

In this subsection, we consider the case where a new character $T[j+1]$ that is appended to the right of the current window $T[i..j]$ already occurs in $T[i..j]$. This means that $\sigma_{i,j} = \sigma_{i,j+1}$, i.e., the alphabet size does not increase before and after the new character is added.

The next lemma shows that a conflict occurs between $\mathcal{M}_A$ and $\mathcal{M}_B$ when $\sigma_{i,j} = \sigma_{i,j+1}$.

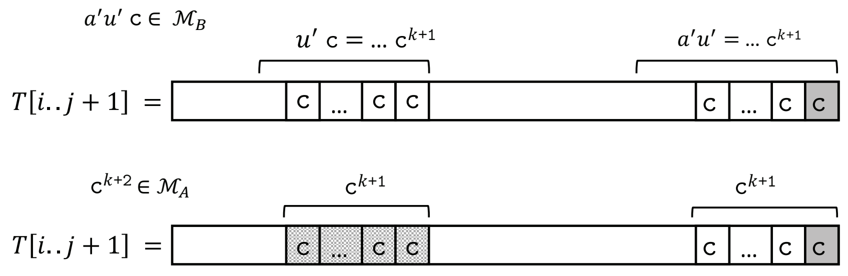


Figure 3.3: Collision between the new Type-B MAW and Type-A MAW, where the rightmost c in gray is the new appended character in each picture.

Lemma 3.6. *For any $T[i..j]$ such that $d = j-i+1 \geq 3$ and $\sigma_{i,j} = \sigma_{i,j+1}$, $|\mathcal{M}_A|+|\mathcal{M}_B| \leq \sigma_{i,j}$.*

Proof. Let $\mathtt{c} = T[j+1]$ and let k be the length of the maximal run of c's that is a suffix of $T[i..j]$. If $T[j] \neq \mathtt{c}$ then let $k = 0$. By the definition of $\mathcal{M}_A$, $\mathtt{c}^{k+2}$ is the only candidate for the Type-A MAW for $T[i..j+1]$, in which case $au = ub = \mathtt{c}^{k+1}$ occurs only once in $T[i..j+1]$ as a suffix. This means that $\mathtt{c}^{k+2}$ can be a Type-A MAW for $T[i..j+1]$ only if $\mathtt{c}^k$ is the longest run of c's in $T[i..j]$.

Now suppose that $\mathtt{c}^{k+2}$ is a Type-A MAW for $T[i..j+1]$, and let $a'u'\mathtt{c}$ denote a Type-B MAW for $T[i..j+1]$. Then, by definition, $a'u'$ occurs only once in $T[i..j+1]$ as a suffix (see also the middle of Figure 3.1).

- If $|u'| \geq k$, then $\mathtt{c}^{k+1}$ is a suffix of u' as shown in Figure 3.3. However, by the definition of Type-B MAWs, $u'\mathtt{c}$ must occur in $T[i..j]$ (see also the middle of Figure 3.1), which implies that $\mathtt{c}^{k+1}$ occurs in $T[i..j]$. This contradicts that $\mathtt{c}^k$ is the longest run of c's in $T[i..j]$.

- If $|u'| < k$, then $a'u'\mathtt{c} = \mathtt{c}^{|a'u'\mathtt{c}|}$ with $|a'u'\mathtt{c}| \leq k+1$ occurs in $T[i..j+1]$ as a suffix, and this contradicts that $a'u'\mathtt{c}$ is a MAW for $T[i..j+1]$.

Hence $a'u'\mathtt{c}$ cannot be in $\mathcal{M}_B$, which leads to $|\mathcal{M}_B| \leq \sigma_{i,j} - 1$ by Lemma 3.3. Thus, $|\mathcal{M}_A| + |\mathcal{M}_B| \leq \sigma_{i,j}$ for any string $T[i..j+1]$ such that $T[i..j]$ contains at least one character that is equal to $T[j+1]$. $\square$

Recall that Lemma 3.2 and Lemma 3.3 in the case where $\sigma_{i,j+1} \geq \sigma_{i,j}$ gives us $|\mathcal{M}_A| + |\mathcal{M}_B| = \sigma_{i,j} + 1$. Compared to this, Lemma 3.6 shaves the total size of $\mathcal{M}_A$ and $\mathcal{M}_B$ by one in the case where $T[j+1]$ already occurs in $T[i..j]$. Coupled with Lemma 3.4, Lemma 3.6 leads us to the following corollary:

Corollary 3.1. *For any* $1 \leq i \leq j < n$, $|\mathsf{MAW}(T[i..j+1]) \triangle \mathsf{MAW}(T[i..j])| \leq \sigma_{i,j+1} + d$, *where* $d = j - i + 1$.

Changes to MAWs when the leftmost character is deleted

Next, we analyze the number of changes of MAWs when deleting the leftmost character from a string. By a symmetric argument to Theorem 3.1, we obtain:

Corollary 3.2. *For any* $1 < i \leq j \leq n$, $|\mathsf{MAW}(T[i..j]) \triangle \mathsf{MAW}(T[i-1..j])| \leq \sigma_{i,j} + d + 1$ *where* $d = j - i + 1$ *and* $\sigma_{i,j}$ *is the number of distinct characters that occur in* $T[i..j]$. *Also, the upper bound is tight when* $\sigma \geq 3$ *and* $\sigma_{i,j} + 1 \leq \sigma$.

Finally, by combining Theorem 3.1 and Corollary 3.2, we obtain the next theorem:

Theorem 3.2. *Let* d *be the window length. For any string* T *of length* $n > d$ *and each position* i *in* T *with* $1 \leq i \leq n-d$, $|\mathsf{MAW}(T[i..i+d-1]) \triangle \mathsf{MAW}(T[i+1..i+d])| \in O(d)$. *Also, there exists a string* T' *with* $|T'| \geq d+1$ *which satisfies* $|\mathsf{MAW}(T'[j..j+d-1]) \triangle \mathsf{MAW}(T'[j+1..j+d])| \in \Omega(d)$ *for some* j *with* $1 \leq j \leq |T'| - d$.

This theorem improves Crochemore et al.'s upper bound for $|\mathsf{MAW}(T[i..i+d-1]) \triangle \mathsf{MAW}(T[i+1..i+d])| \in O(\sigma d)$ for any alphabet of size $\sigma \in \omega(1)$.

Total changes of MAWs when sliding a window on a string

In this subsection, we consider the total number of changes of MAWs when sliding the window of length d from the beginning of T to the end of T. We denote the total number of changes of MAWs by $\mathcal{S}(T,d) = \sum_{i=1}^{n-d} |\mathsf{MAW}(T[i..i+d-1]) \triangle \mathsf{MAW}(T[i+1..i+d])|$. The following lemma is known:

Lemma 3.7 ([30])**.** *For a string* T *of length* $n > d$ *over an alphabet* Σ *of size* σ, $\mathcal{S}(T,d) \in O(\sigma n)$.

The aim of this subsection is to give a more rigorous bound for $\mathcal{S}(T,d)$. We first show that the above bound is tight under some conditions.

Lemma 3.8. *The upper bound of Lemma 3.7 is tight when* $\sigma \leq d$ *and* $n - d \in \Omega(n)$.

Proof. If $\sigma = 2$, the lower bound $\mathcal{S}(T,d) \in \Omega(n-d) = \Omega(\sigma(n-d))$ is obtained by string $T = (\mathtt{ab})^{n/2}$ since $\mathsf{MAW}((\mathtt{ab})^{d/2}) \triangle \mathsf{MAW}((\mathtt{ba})^{d/2}) = \{(\mathtt{ab})^{d/2}, (\mathtt{ba})^{d/2}\}$.

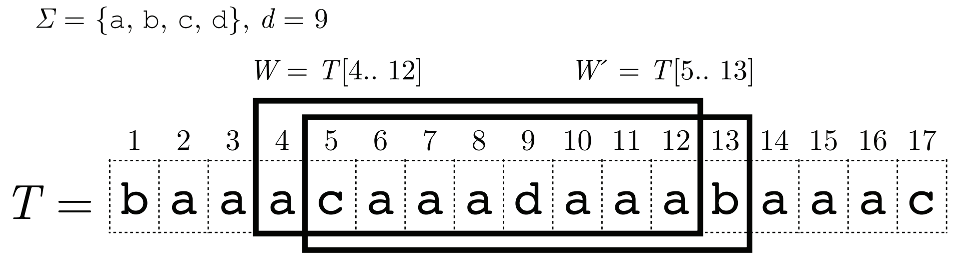
In the sequel, we consider the case where $\sigma \geq 3$. Let k be the integer with $(k-1)(\sigma-1) \leq d < k(\sigma-1)$. Note that $k \geq 2$ since $\sigma \leq d$. Let $\Sigma = \{a_1, a_2, \cdots, a_\sigma\}$ and $\alpha = a_\sigma$. We consider a string $T' = U^e + U[..m]$ where $U = a_1\alpha^{k-1}a_2\alpha^{k-1}\ldots a_{\sigma-1}\alpha^{k-1}$, $e = \lfloor \frac{n}{k(\sigma-1)} \rfloor$, and $m = n \bmod k(\sigma-1)$. Let c be a character that is not equal to α. For any two distinct occurrences $i_1, i_2 \in occ_{T'}(c)$ for c, $|i_1 - i_2| \geq k(\sigma-1) > d$. Thus, any character $c \neq \alpha$ is absent from at least one of two adjacent windows $T'[i..i+d-1]$ and $T'[i+1..i+d]$ for every $1 \leq i \leq n-d$.

Now we consider a window $W = T'[p-d..p-1]$ where $d+1 \leq p \leq n$ and $T'[p] = \beta \neq \alpha$. Let $\Pi = \{b_1, b_2, \cdots, b_{\pi-1}, \alpha\} \subset \Sigma \setminus \{\beta\}$ be the set of all π characters that occur in W. Without loss of generality, we assume that the current window is $W = \alpha^r b_1\alpha^{k-1}b_2\alpha^{k-1}\cdots b_{\pi-1}\alpha^{k-1}$ and the next window is $W' = W[2..]\beta$ where $r = d \bmod k$ (see Figure 3.4). For any character $b \in \Pi \setminus \{b_1, b_{\pi-1}, \alpha\}$, $b\alpha^\ell\beta$ is in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$ for every $0 \leq \ell \leq k-1$. If $r > 0$, $b_1\alpha^\ell\beta$ is also in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$ for every $0 \leq \ell \leq k-1$. Otherwise, b_1 is in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$ and $b_1\alpha^\ell b_2$ is in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$ for every $0 \leq \ell \leq k-2$ since b_1 is absent from W'. Also, β is in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$ and $b_{\pi-1}\alpha^\ell\beta$ is in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$ for every $0 \leq \ell \leq k-2$. Thus, at least $(\pi-3)k + k + 1 + (k-1) = (\pi-1)k$ MAWs are in $\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)$. Additionally, the number $\pi - 1$ of distinct characters which occur in W and are not equal to α is at least $\lfloor(\sigma-1)/2\rfloor$, since $k\lfloor(\sigma-1)/2\rfloor \leq k(\sigma-1)/2 = (k-k/2)(\sigma-1) \leq (k-1)(\sigma-1) \leq d$ where the second inequality follows from $k \geq 2$. Therefore, $|\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)| \geq (\pi-1)k \geq \lfloor(\sigma-1)/2\rfloor k \in \Omega(\sigma k) = \Omega(d)$. The number of pairs of two adjacent windows W and W' where $|\mathsf{MAW}(W')\triangle\mathsf{MAW}(W)| \in \Omega(d)$ is $\Theta((n-d)/k)$. Therefore, we obtain $\mathcal{S}(T', d) \in \Omega(d(n-d)/k) = \Omega(\sigma(n-d)) = \Omega(\sigma n)$ since $n - d \in \Omega(n)$. $\square$

Next, we consider the case where $\sigma \geq d+1$.

Lemma 3.9. *For a string T of length $n > d$ over an alphabet Σ of size σ, $\mathcal{S}(T, d) \in O(d(n-d))$, and this upper bound is tight when $\sigma \geq d+1$.*

Proof. By Theorem 3.2, it is clear that $\mathcal{S}(T, d) \in O(d(n-d))$. Next, we show that there is a string T' of length $n > d$ such that $\mathcal{S}(T', d) \in \Omega(d(n-d))$ for any integer d with $1 \leq d \leq \sigma-1$. Let $\Sigma = \{a_1, a_2, \cdots, a_\sigma\}$. We consider a string $T' = (a_1a_2\cdots a_{d+1})^e a_1a_2\cdots a_k$ where $e = \lfloor n/(d+1)\rfloor$ and $k = n \bmod (d+1)$. For each window $W = T'[i..i+d-1]$ in T', W consists of distinct d characters, and the character $T'[i+d]$ that is the right neighbor of W is different from any characters that occur in W. Without loss of generality , we assume that the current window is $W = a_1a_2\cdots a_d$ and the next window is $W' = W[2..]a_{d+1} =$


$\Sigma = \{$a, b, c, d$\}$, $d = 9$
$W = T[4..\ 12]$
$W' = T[5..\ 13]$
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
$T =$ b a a a c a a a d a a a b a a a c

$\mathsf{MAW}(W) = \{$aaaa, cc, cd, cad, caad, dd, dad, daad, daaad, dc, b, aac, cac, dac$\}$
$\mathsf{MAW}(W') = \{$aaaa, cc, cd, cad, caad, dd, dad, daad, daaad, dc, ac, ba, bb, bc, bd, cb, cab, caab, caaab, db, dab, daab$\}$

Figure 3.4: Illustration of examples of MAWs for adjacent two windows. In this example, $\sigma = 4, d = 9$, and $k = 4$. The size of the symmetric difference of $\mathsf{MAW}(W)$ and $\mathsf{MAW}(W')$ is $|\mathsf{MAW}(W) \triangle \mathsf{MAW}(W')| = |\{$b, aac, cac, dac, ac, ba, bb, bc, bd, cb, cab, caab, caaab, db, dab, daab$\}| = 16$.

$a_2 \cdots a_{d+1}$. Then, $|\mathsf{MAW}(W') \triangle \mathsf{MAW}(W)| = |\{{a_1}^2, a_{d+1}, a_2a_1, \ldots, a_da_1, a_1a_3, \ldots, a_1a_d\} \cup \{a_1, {a_{d+1}}^2, a_{d+1}a_2, \ldots, a_{d+1}a_d, a_2a_{d+1}, \ldots, a_{d-1}a_{d+1}\}| = 4d - 2 \in \Omega(d)$. Therefore, $\mathcal{S}(T', d) = \Omega(d(n-d))$. $\square$

The main result of this section follows from the above lemmas:

Theorem 3.3. *For a string T of length $n > d$ over an alphabet Σ of size σ, $\mathcal{S}(T, d) \in O(\min\{d, \sigma\}n)$. This upper bound is tight when $n - d \in \Omega(n)$.*

We remark that $n - d \in \Omega(n)$ covers most interesting cases for the window length d, since the value of d can range from $O(1)$ to cn for any $0 < c < 1$.

3.1.2 Tighter bounds for binary alphabets

In this section we consider the case where $\sigma' = 2$, i.e. when both the current sliding window $S = T[i..i+d-1]$ and the next window $S\alpha = T[i..i+d]$ extended with a new character $\alpha = T[i+d]$ consist of two distinct characters. The goal of this section is to show that when $\sigma' = 2$, there exists a tighter upper bound for the number of changes of MAWs than the general case with $\sigma' \geq 3$.

In what follows, let us denote by $\Sigma_2 = \{\mathtt{0}, \mathtt{1}\}$ the binary alphabet, and assume without loss of generality that we append the new character $\alpha = \mathtt{0}$ to the window S of length d and obtain the extended window $S\alpha = S\mathtt{0}$.

As a warm up, we begin with the two following lemmas which show that at most 3 MAWs can change in the cases where $d = 1$ and $d = 2$ for any binary strings.

Lemma 3.10. *For any string S over Σ_2 with $|S| = d = 1$, $|\mathsf{MAW}(S) \triangle \mathsf{MAW}(S\mathtt{0})| \leq 3$.*

Proof. For each $S \in \{\mathtt{0}, \mathtt{1}\}$ of length 1,

$$\begin{aligned} \mathsf{MAW}(\mathtt{0}) \triangle \mathsf{MAW}(\mathtt{00}) &= \{\underline{\mathtt{00}}, \mathtt{000}\}, \\ \mathsf{MAW}(\mathtt{1}) \triangle \mathsf{MAW}(\mathtt{10}) &= \{\underline{\mathtt{0}}, \mathtt{00}, \mathtt{01}\}, \end{aligned}$$

where the underlined strings are those in $\mathsf{MAW}(S) \setminus \mathsf{MAW}(S\mathtt{0})$ and the strings without underlines are those in $\mathsf{MAW}(S\mathtt{0}) \setminus \mathsf{MAW}(S)$. Thus the lemma holds. □

Lemma 3.11. *For any string S over Σ_2 with $|S| = d = 2$, $|\mathsf{MAW}(S) \triangle \mathsf{MAW}(S\mathtt{0})| \leq 3$.*

Proof. For each $S \in \{\mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}\}$ of length 2,

$$\begin{aligned} \mathsf{MAW}(\mathtt{00}) \triangle \mathsf{MAW}(\mathtt{000}) &= \{\underline{\mathtt{000}}, \mathtt{0000}\}, \\ \mathsf{MAW}(\mathtt{01}) \triangle \mathsf{MAW}(\mathtt{010}) &= \{\underline{\mathtt{10}}, \mathtt{101}\}, \\ \mathsf{MAW}(\mathtt{10}) \triangle \mathsf{MAW}(\mathtt{100}) &= \{\underline{\mathtt{00}}, \mathtt{000}\}, \\ \mathsf{MAW}(\mathtt{11}) \triangle \mathsf{MAW}(\mathtt{110}) &= \{\underline{\mathtt{0}}, \mathtt{00}, \mathtt{01}\}, \end{aligned}$$

where the underlined strings are those in $\mathsf{MAW}(S) \setminus \mathsf{MAW}(S\mathtt{0})$ and the strings without underlines are those in $\mathsf{MAW}(S\mathtt{0}) \setminus \mathsf{MAW}(S)$. Thus the lemma holds. □

We move onto the case where $d \geq 3$. Our first observation is that it is sufficient to consider the case that S is not unary. For any d, it is clear that $|\mathsf{MAW}(\mathtt{0}^d) \triangle \mathsf{MAW}(\mathtt{0}^{d+1})| = 2$. Now let us consider $\mathtt{1}^d$ in the next lemma.

Lemma 3.12. *For any $d \geq 3$ let $V = \mathtt{1}^d$. Then, there exists another string S of length d over Σ_2 such that $S[k] = \mathtt{0}$ for some $1 \leq k \leq d$ and $|\mathsf{MAW}(V) \triangle \mathsf{MAW}(V\mathtt{0})| \leq |\mathsf{MAW}(S) \triangle \mathsf{MAW}(S\mathtt{0})|$.*

Proof. Since $V = \mathtt{1}^d$, $\mathsf{MAW}(V) \setminus \mathsf{MAW}(V\mathtt{0}) = \{\mathtt{0}\}$. Also, $\mathsf{MAW}(V\mathtt{0}) \setminus \mathsf{MAW}(V) = \{\mathtt{00}, \mathtt{01}\}$. Thus $|\mathsf{MAW}(V) \triangle \mathsf{MAW}(V\mathtt{0})| = 3$ for any $d \geq 1$.

Let $S = \mathtt{0}\mathtt{1}^{d-1}$ and $S\mathtt{0} = \mathtt{0}\mathtt{1}^{d-1}\mathtt{0}$ with $d \geq 3$. Then, $\mathsf{MAW}(S\mathtt{0}) \setminus \mathsf{MAW}(S) = \{\mathtt{0}\mathtt{1}^k\mathtt{0} \mid 1 \leq k \leq d-2\} \cup \{\mathtt{101}\}$ and $\mathsf{MAW}(S\mathtt{0}) \setminus \mathsf{MAW}(S) = \{\mathtt{10}\}$. Thus we have $|\mathsf{MAW}(S) \triangle \mathsf{MAW}(S\mathtt{0})| \geq d \geq 3$. □

According to Lemmas 3.10, 3.11 and 3.12, in what follows we focus on the case where $d \geq 3$ and the current window $S = T[i..i+d-1]$ contains at least one $\mathtt{0}$. The latter condition implies that we focus on the case where the new character $\alpha = \mathtt{0}$ already occurs in the window S.

As in the case of non-binary alphabets, we analyze the numbers of added Type-A/Type-B/Type-C MAWs in $\mathcal{M}_A$/$\mathcal{M}_B$/$\mathcal{M}_C$ for binary strings. Recall that in the current context, for any $S = T[i..i+d-1]$, a MAW w in $\mathsf{MAW}(S\mathtt{0}) \setminus \mathsf{MAW}(S)$ is said to be of:

- Type-A if neither $w[2..]$ nor $w[..|w|-1]$ occurs in S;
- Type-B if $w[2..]$ occurs in S but $w[..|w|-1]$ does not occur in S;
- Type-C if $w[2..]$ is does not occur in S but $w[..|w|-1]$ occurs in S.

We first show the upper bound for the size of $\mathcal{M}_C$ in the case where $\sigma' = 2$.

Lemma 3.13. *For any binary string S over Σ_2 such that $|S| = d \geq 3$, $|\mathcal{M}_C| \leq d-2$.*

Proof. Recall the proof for Lemma 3.4. There, we proved that each MAW w of Type-C for any non-binary string $R\alpha = T[i..i+d] = T[i..j+1]$ is mapped by an injection f to a distinct position of $T[i..j]$ in the range $[i, j-1]$, or alternatively to a distinct position of R in range $[1, d-1]$. This showed $|\mathcal{M}_C| \leq d-1$ for $\sigma' \geq 3$.

Here we show that the range of such an injection f is $[2, d-1]$ for any binary string S with $\sigma' = 2$. Since the appended character is $\alpha = \mathtt{0}$, and since the candidate x for the MAW of Type-C which should be mapped to the first position in S is of length 2, the candidate x has to be either $\mathtt{00}$ or $\mathtt{10}$.

(1) If $x = \mathtt{00}$, then $S[1] = \mathtt{0}$. If $\mathtt{00}$ does not occur in S (see also the top picture of Figure 3.5), then $\mathtt{00}$ is already a MAW for S (i.e. $\mathtt{00} \in \mathsf{MAW}(S)$). Thus $\mathtt{00} \notin \mathsf{MAW}(S\mathtt{0}) \setminus \mathsf{MAW}(S)$ in this case. Otherwise ($\mathtt{00}$ occurs in S), then clearly $\mathtt{00}$ is not a MAW for $S\mathtt{0}$ (see also the middle picture of Figure 3.5).

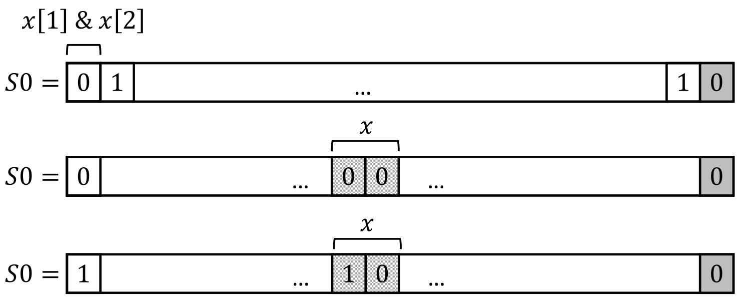


Figure 3.5: Characteristics of Type-C MAWs in the binary case with $\sigma' = 2$, where the right-most 0 in gray is the new appended character in each picture.

(2) If $x = 10$, then $S[1] = 1$. However, since the appended character is 0, 10 must occur somewhere in $S0$ (see also the bottom picture of Figure 3.5). Thus 10 is not a MAW for $S0$.

Hence, the first position of S cannot be assigned to any MAW of Type-C for $S0$, leading to $|\mathcal{M}_C| \leq d - 2$ for any binary string S of length $d \geq 3$. □

In other words, Lemma 3.13 shows that in the binary case with $\sigma' = 2$, the maximum number of added Type-C MAWs is 1 less than in the case with $\sigma' \geq 3$.

Next, we consider the total number of added Type-A/Type-B MAWs.

From Lemma 3.6, the next corollary holds.

Corollary 3.3. $|\mathcal{M}_A| + |\mathcal{M}_B| \leq 2$ *on any binary string S when the character to be appended already occurs in S.*

A direct consequence of Lemma 3.13 and Lemma 3.6 is an upper bound for the added MAWs $|\mathcal{M}_A| + |\mathcal{M}_B| + |\mathcal{M}_C| \leq d$ for any binary string $S0$ with $|S| = d \geq 3$. In what follows, we further reduce this upper bound to $|\mathcal{M}_A| + |\mathcal{M}_B| + |\mathcal{M}_C| \leq d - 1$. For this purpose, we introduce the next lemma:

Lemma 3.14. *For any binary string S over Σ_2 such that $|S| = d \geq 3$, $|\mathcal{M}_B|$ is at most the number of occurrences of* 0 *in $S[1..d-1]$, and $|\mathcal{M}_C|$ is at most the number of occurrences of* 1 *in $S[3..d]$.*

Proof. First we consider Type-B MAWs for $S0$. Since S is a binary string, by Lemma 3.3, there are at most two MAWs in $\mathcal{M}_B$. We assume that there are two MAWs in $\mathcal{M}_B$ and let $au0$ and $a'1$ be the two MAWs where $a, a' \in \Sigma_2$ and $u, u' \in \Sigma_2^*$. By the definition of Type-B MAWs, $u0$ and $u'1$ occur in S and $u[|u|] = u'[u'] = 0$ since au and $a'u'$ occur in $S0$ as suffixes. For any two positions t_0 and t_1 such that $S[t_0..t_0+|u\mathtt{0}|-1] = u\mathtt{0}$, $S[t_1..t_1+|u'\mathtt{1}|-1] = u'\mathtt{1}$, $t_0+|u| \neq t_1+|u'|$. Consequently, there are at least two occurrences of $\mathtt{0}$ in S since $S[t_0+|u|-1] = \mathtt{0}$ and $S[t_1+|u'|-1] = \mathtt{0}$. Since $t_0+|u\mathtt{0}|-1 \leq d$, $|\mathcal{M}_B|$ is at most the number of occurrences of $\mathtt{0}$ in $S[1..d-1]$.

Second we consider Type-C MAWs for $S\mathtt{0}$. let $au\mathtt{0}$ be the Type-C MAW where $a \in \Sigma_2$, $u, \in \Sigma_2^*$ since $u\mathtt{0}$ must be a suffix of $S\mathtt{0}$. By the definition of Type-C MAW, there has to be an occurrence of au in S. Note that this occurrence has to be immediately followed by a $\mathtt{1}$ since $au\mathtt{0}$ does not occur in $S\mathtt{0}$. Thus, for each $au\mathtt{0} \in \mathcal{M}_C$, we need an occurrence of $au\mathtt{1}$ in S. Since $|au| \geq 1$, we clearly cannot use the first position of S as the ending position of $au1$. Also, it follows from Lemma 3.13 (and its proof) that the second position of S cannot be the ending position of au for any Type-C MAW aub for $S\mathtt{0}$. This implies that there is no Type-C MAW that corresponds to the $\mathtt{1}$ in the second position of S. Thus, the total number of Type-C MAWs for $S\mathtt{0}$ is upper bounded by the number of occurrences of $\mathtt{1}$ in $S[3..d]$. $\square$

Intuitively, Lemma 3.14 implies that flipping substrings $\mathtt{1}$ in $S[3..d-1]$ does not increase the total number of Type-B and Type-C MAWs for $S\mathtt{0}$.

Lemma 3.15. *For any binary string S over Σ_2 with $|S| = d \geq 3$, $|\mathcal{M}_A|+|\mathcal{M}_B|+|\mathcal{M}_C| \leq d-1$.*

Proof. S is not unary due to Lemma 3.12. It immediately follows from Lemma 3.13 and Corollary 3.3 that $|\mathcal{M}_A|+|\mathcal{M}_B|+|\mathcal{M}_C| \leq d$, and assume on the contrary that there exists a binary string S' over Σ_2 such that $|\mathcal{M}_A|+|\mathcal{M}_B|+|\mathcal{M}_C| = d$. Then, it has to be $|\mathcal{M}_A|+|\mathcal{M}_B| = 2$ and $|\mathcal{M}_C| = d-2$, again by Lemma 3.13 and Corollary 3.3. Therefore, $S'[3..d] = \mathtt{1}^{d-2}$ by Lemma 3.14, and $|\mathcal{M}_A| = 0$. Hence we must have $|\mathcal{M}_B| = 2$, which leads to $S' = \mathtt{001}^{d-2}$. Now the sets of all the added MAWs for $S'\mathtt{0} = \mathtt{001}^{d-2}\mathtt{0}$ are

$$\begin{aligned}
\mathcal{M}_A &= \emptyset, \\
\mathcal{M}_B &= \{\mathtt{100}, \mathtt{101}\}, \\
\mathcal{M}_C &= \{\mathtt{01}^k\mathtt{0} \mid 1 \leq k \leq d-3\},
\end{aligned}$$

which leads to $|\mathcal{M}_A|+|\mathcal{M}_B|+|\mathcal{M}_C| = d-1$, a contradiction. Thus the lemma holds. $\square$

We obtain our main theorem:

Theorem 3.4. *For any binary string S over Σ_2 with $|S| = d \geq 3$, $|\mathsf{MAW}(S) \triangle \mathsf{MAW}(S\mathtt{0})| \leq d$, and this upper bound is tight.*

Proof. The upper bound follows from Lemmas 3.1 and 3.15, and its tightness follows from and our construction of the string $S'\mathtt{0}$ in the proof for Lemma 3.15. $\square$

The next corollary summarizes the results of this section.

Corollary 3.4. *For any binary string S over Σ_2 with $|S| = d$, $|\mathsf{MAW}(S) \triangle \mathsf{MAW}(S\mathtt{0})| \leq \max\{3, d\}$, and this upper bound is tight for any $d \geq 1$.*

Proof. The upper bound follows from Lemmas 3.10 and 3.11, Theorem 3.4, and its tightness follows from all possible binary cases shown in the proof for Lemmas 3.10 and 3.11 for $d = \{1, 2\}$, and our construction of the string $S'\mathtt{0}$ in the proof for Lemma 3.15 for $d \geq 3$. $\square$

3.2 MAWs for RLE strings

In this section, we present our new bounds for the changes of MAWs for the Run-Length Encoded (RLE) string T. We begin with lower bounds of Type-2,4,5 MAWs in Section 3.2.1. After that, we show the upper bounds of Type-2,4,5 MAWs in Section 3.2.2. Note that Type-1 MAWs lower bound is omitted since we show the tight number of Type-1 MAWs in Section 3.2.2. Type-3 MAW lower bound is also omitted since it is unbounded by the RLE size m, which we show in Section 3.2.2. In Section 3.2.4, we give the compact data structures which can efficiently output each type MAWs for T upon query. Finally, in Section 3.2.5, we explain how to construct the representation shown in Section 3.2.4 in $O(m \log m)$ time.

3.2.1 Lower bounds on the number of MAWs for RLE strings

In this section, we give tight lower bounds for the sizes of $\mathcal{M}_2$, $\mathcal{M}_4$, and $\mathcal{M}_5$ which asymptotically match the upper bounds given in section 3.2.2. For Type-1 and Type-3 MAWs, we show a tight bound $|\mathcal{M}_1| = \sigma$ and the fact that $|\mathcal{M}_3|$ is unbounded by the RLE size m in section 3.2.2. Throughout this section, we omit the terminal \$ at either end of T, since our lower bound instances do not need them.

Lemma 3.16. *There exists a string T such that $|\mathcal{M}_2| = \sigma'(\sigma' - 2) + 1$.*

Proof. Let $T = \mathtt{123} \cdots \sigma'$, where all characters in T are mutually distinct. Any bigram occurring in T is of the form $i(i+1)$ with $1 \leq i < \sigma'$. Thus, for each $1 \leq i < \sigma'$, bigram $i \cdot j$ with any $j \in \{1, \ldots, i-1, i+2, \ldots, \sigma'\}$ is a Type-2 MAW for T, and bigram $\sigma' \cdot j$ is a Type-2 MAW for T. Namely, the set $\mathcal{M}_2$ of Type-2 MAWs for T is:

$$\mathcal{M}_2 = \left\{ \begin{array}{l} \mathtt{13}, \ldots, \mathtt{1}\sigma', \\ \mathtt{21}, \mathtt{24}, \ldots, \mathtt{2}\sigma', \\ \mathtt{31}, \mathtt{32}, \mathtt{35}, \ldots, \mathtt{3}\sigma', \\ \ldots, \\ (\sigma'-1)\mathtt{1}, \ldots, (\sigma'-1)(\sigma'-2), \\ \sigma'\mathtt{1}, \ldots, \sigma'(\sigma'-1) \end{array} \right\}.$$

Thus we have $|\mathcal{M}_2| = \sigma'(\sigma' - 2) + 1$ for this string T. □

Since $\sigma' = m$ for the string T of Lemma 3.16, we obtain a tight lower bound $|\mathcal{M}_2| \in \Omega(m^2)$ in terms of m. The string $T = \mathtt{123} \cdots \sigma'$ can easily be generalized so that $m < n$, where

$n = |T|$. For instance, consider $T' = 1^{p_1}2^{p_2}3^{p_3}\cdots\sigma'^{p_{\sigma'}}$ with $p_i > 1$ for each i. The set of Type-2 MAWs for T' is equal to that for T.

Lemma 3.17. *There exists a string T with $R(T) = m$ such that $|\mathcal{M}_4| \in \Omega(m^2)$.*

Proof. Consider string $T = \mathtt{abc}^p \cdot \mathtt{ab}^2\mathtt{c}^{p-1} \cdot \mathtt{ab}^3\mathtt{c}^{p-2} \cdot \mathtt{ab}^4\mathtt{c}^{p-3} \cdots \mathtt{ab}^{p-1}\mathtt{c}^2 \cdot \mathtt{ab}^p\mathtt{c} \cdot \mathtt{a}$, where $\mathtt{a}$, $\mathtt{b}$, and $\mathtt{c}$ are mutually distinct characters.

Then the set of Type-4 MAWs for T is a superset of the following set:

$$\left\{\begin{array}{l} \mathtt{abca}, \mathtt{abc}^2\mathtt{a}, \ldots, \mathtt{abc}^{p-1}\mathtt{a}, \\ \mathtt{ab}^2\mathtt{ca}, \mathtt{ab}^2\mathtt{c}^2\mathtt{a}, \ldots, \mathtt{ab}^2\mathtt{c}^{p-2}\mathtt{a}, \\ \mathtt{ab}^3\mathtt{ca}, \mathtt{ab}^3\mathtt{c}^2\mathtt{a}, \ldots, \mathtt{ab}^3\mathtt{c}^{p-3}\mathtt{a}, \\ \ldots, \\ \mathtt{ab}^{p-2}\mathtt{ca}, \mathtt{ab}^{p-2}\mathtt{c}^2\mathtt{a}, \\ \mathtt{ab}^{p-1}\mathtt{ca} \end{array}\right\}.$$

Since $m = 3p + 1$, we have $|\mathcal{M}_4| > p(p-1)/2 \in \Omega(p^2) = \Omega(m^2)$. $\square$

Lemma 3.18. *There exists a string T with $R(T) = m$ such that $|\mathcal{M}_5| \in \Omega(m)$.*

Proof. Consider string $T = \mathtt{abc} \cdot \mathtt{ab}^2\mathtt{c}^2 \cdot \mathtt{ab}^3\mathtt{c}^3 \cdots \mathtt{ab}^p\mathtt{c}^p \cdot \mathtt{a}$, where $\mathtt{a}$, $\mathtt{b}$, and $\mathtt{c}$ are mutually distinct characters. Then the set of Type-5 MAWs for T is a superset of the set

$$\{\mathtt{b}^{i+1}\mathtt{c}^i\mathtt{a} \mid 1 \le i \le p-1\}.$$

Since $m = 3p + 1$, $|\mathcal{M}_5| > p - 1 \in \Omega(p) = \Omega(m)$. $\square$

3.2.2 Upper bounds on the number of MAWs for RLE strings

In this section, we present upper bounds for the number of MAWs in a string T that is represented by its RLE $\mathsf{rle}(T)$ of size $R(T) = m$.

Upper bounds for the number of MAWs of Type-1, 2, 3, 5

We first consider the number of MAWs except for those of Type-4.

Lemma 3.19. $|\mathcal{M}_1| = \sigma$.

Proof. By the definition of $\mathcal{M}_1$, any MAW in $\mathcal{M}_1$ is of the form a^k. For any character $\alpha \in \Sigma'$ that occurs in T, let $aub = \alpha^{p+1}$ such that α^p is the *longest* maximal run of α in T. Clearly $\alpha^p = au = ub$ occurs in T and α^{p+1} does not occur in T. Since $R(aub) = R(\alpha^{p+1}) = 1$, $\alpha^{p+1} \in \mathcal{M}_1$ and it is the unique MAW of Type-1 consisting of α's. For any character $\beta \in \Sigma \setminus \Sigma'$ that does not occur in T, clearly β is a MAW of T and $\beta \in \mathcal{M}_1$ since $R(\beta) = 1$. In total, we obtain $|\mathcal{M}_1| = \sigma$. $\square$

Note that this upper bound for $|\mathcal{M}_1|$ is tight for any string T and alphabet Σ of size σ.

Lemma 3.20. $|\mathcal{M}_2| \in O((\sigma')^2)$.

Proof. Any MAW in $\mathcal{M}_2$ is of the form ab with $a, b \in \Sigma$ and $a \neq b$. By the definition of MAWs, ab can be a MAW for T only if both a and b occur in T, which implies that $a, b \in \Sigma'$. The number of such combinations of a and b is $\sigma'(\sigma' - 1)$. $\square$

Since $\sigma' \leq m$ always holds, we have that $|\mathcal{M}_2| \in O(m^2)$. Later we will show that this upper bound for $|\mathcal{M}_2|$ is asymptotically tight.

Lemma 3.21. $|\mathcal{M}_3|$ *is unbounded by* m.

Proof. Consider a string $T = ac^{n-2}b$, where $a \neq c$ and $c \neq b$. Then ac^kb for each $1 \leq k \leq n-3$ is a MAW of T and $R(ac^kb) = 3$. Since they are the only Type-3 MAWs of T, we have that $|\mathcal{M}_3| = n - 3$. Clearly, the original length n of T cannot be bounded by $m = R(T) = 3$. $\square$

While Lemma 3.21 is enough to show that $|\mathcal{M}_3|$ can be linear in n and is unbounded by m, the number of MAWs in a string is known to be $O(\sigma n)$ [30]. Indeed, we have a stronger result than Lemma 3.21, namely, there is a string T such that the number of Type-3 MAWs in T is $\Omega(\sigma n)$. Let p be an integer which satisfies that $n = 1 + (p+1)(\sigma - 1)$, and a_i be the i-th character in Σ. Then,

$$T = a_2 a_1^p \cdot a_3 a_1^p \cdots a_{\sigma-1} a_1^p \cdot a_\sigma \qquad (3.1)$$

satisfies that the number of Type-3 MAWs in T is $\Omega(\sigma n)$. For any triplet j_1, j_2, j_3 with $1 \leq j_1 < \sigma, 1 < j_1 \leq \sigma, 0 \leq j_2 \leq p, j_1 \neq j_3 - 1$, obviously $a_{j_1} a_1^{j_2} a_{j_3}$ is a Type-3 MAW of T, which is at least $\sigma \cdot p \cdot \sigma = \Omega(\sigma n)$.

Although the number of MAWs of Type-3 is unbounded by m, later we will present an $O(m)$-space data structure that can enumerate all elements in $\mathcal{M}_3$ in output-sensitive time.

Lemma 3.22. $|\mathcal{M}_5| \in O(m)$.

Proof. Any MAW $aub \in \mathcal{M}_5$ can be represented by $a^{i+1}vb$ or avb^{i+1} with maximal integer $i \geq 1$, where $a^i v = u$ in the former and $vb^i = u$ in the latter. Let us consider the case of $a^{i+1}vb$ as the case of avb^{i+1} is symmetric. Then $ca^i vb$ with some character $c \neq a$ must occur in T. Let k be the beginning position of an occurrence of $ca^i vb$ in T. Then, $T[k+1..k+i] = a^i$ is a maximal run of a.

Now consider any distinct MAW $a^{i+1}v'b' \in \mathcal{M}_5 \setminus \{a^{i+1}vb\}$ with $v'b' \neq vb$. Again, $c'a^i v'b'$ with some character $c' \neq a$ must occur in T. Suppose on the contrary that $c'a^i v'b'$ has an occurrence beginning at the same position k as $ca^i vb$. This implies that $c' = c$, and both $a^i vb$ and $a^i v'b'$ are prefixes of $T[k+1..|T|]$.

- If $|a^i vb| < |a^i v'b'|$, then $a^i v'$ contains $a^i vb$ as a substring. Since $a^{i+1}v'$ occurs in T, $a^{i+1}vb$ must also occur in T. Hence $a^{i+1}vb$ is not a MAW for T, a contradiction.

- If $|a^i vb| > |a^i v'b'|$, then $a^i v$ contains $a^i v'b'$ as a substring. Thus $a^{i+1}vb$ is an absent word for T but it is not minimal. Hence $a^{i+1}vb$ is not a MAW for T, a contradiction.

- If $|a^i vb| = |a^i v'b'|$, then this contradicts that $a^i ub \neq a^i u'b'$.

Hence, at most two element of $\mathcal{M}_5$ can be associated with a position k in T such that $T[k] \neq T[k+1]$. The number of such positions does not exceed $2m$.

□

3.2.3 Upper bound for the number of MAWs of Type-4

In the rest of this section, we show an upper bound of the number of MAWs of Type-4. Namely, we prove the following lemma.

Lemma 3.23. $|\mathcal{M}_4| \in O(m^2)$.

Firstly, we explain a way to characterize MAWs of Type-4.

For any string $w \in \Sigma^*$ and integer $t > 0$, let $\mathsf{Exp}^t(w)$ be the set of bridges such that $\mathsf{Exp}^t(w) = \{w' \in \mathcal{B} \mid \mathsf{shk}^t(w') = w\}$. Namely, $\mathsf{Exp}^t(w)$ is the *inverse image* of $\mathsf{shk}^t(w') = w$ for bridge substrings w' of T. We use $\mathsf{Exp}(w)$ to denote $\mathsf{Exp}^1(w)$. Figure 3.6 gives an example for $\mathsf{Exp}^t(w)$ ($\mathsf{Exp}^t_+(w)$ in the figure will be defined later). Any MAW z in $\mathcal{M}_4$ is of the form

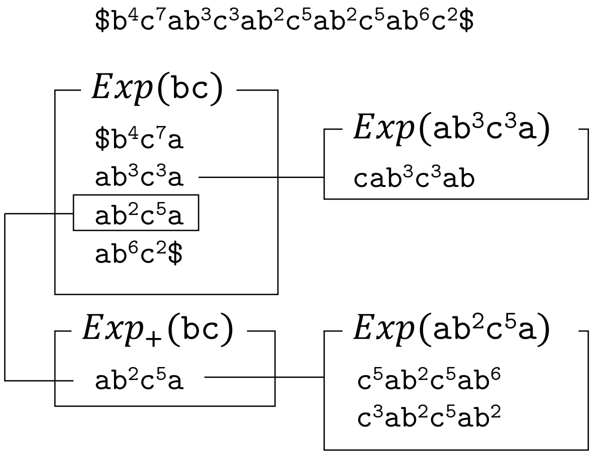


Figure 3.6: The bridge $w_1 = \mathtt{ab^2c^5a} \in \mathsf{Exp}(\mathtt{bc})$ is an element of $\mathsf{Exp}_+(\mathtt{bc})$ since $|\mathsf{Exp}(w_1)| \geq 2$. On the other hand, the bridge $w_2 = \mathtt{ab^3c^3a} \in \mathsf{Exp}(\mathtt{bc})$ is not an element of $\mathsf{Exp}_+(\mathtt{bc})$ since $|\mathsf{Exp}(w_2)| < 2$.

$a\alpha^i u\beta^j b$ with $a, b, \alpha, \beta \in \Sigma$, $u \in \Sigma^*$, and positive integers i, j where a, α^i, β^j, b are the first, the second, the second last, and the last run of z, respectively. By the definition of MAWs, both the suffix $\alpha^i u\beta^j b$ and the prefix $a\alpha^i u\beta^j$ of z occur in T. From this fact, we can obtain the following observations.

Observation 3.1. *Each MAW $z \in \mathcal{M}_4$ corresponds to a pair of distinct bridges $(w_1, w_2) \in \mathsf{Exp}(\mathsf{shk}(z)) \times \mathsf{Exp}(\mathsf{shk}(z))$. Formally, for each MAW $z = a\alpha^i u\beta^j b \in \mathcal{M}_4$, there exist characters $a_1, b_1 \in \Sigma \cup \{\$\}$ and integers $i_1 \geq i, j_1 \geq j$ such that $w_1 = a_1\alpha^{i_1} u\beta^j b, w_2 = a\alpha^i u\beta^{j_1} b_1 \in \mathsf{Exp}(\mathsf{shk}(z))$ and $w_1 \neq w_2$ (since these two occur in T but z does not occur in T).*

This observation gives a main idea of our characterization which is stated in the following lemma.

Lemma 3.24. *For any bridge w, $|\{z \mid \mathsf{shk}(z) = w, z \in \mathcal{M}_4\}| \leq |\mathsf{Exp}(w)|(|\mathsf{Exp}(w)| - 1)$.*

Proof. Let $\mathcal{M}_4(w) = \{z \mid \mathsf{shk}(z) = w, z \in \mathcal{M}_4\}$. By Observation 3.1, each $z \in \mathcal{M}_4(w)$ corresponds to a pair $(w_1, w_2) \in \mathsf{Exp}(\mathsf{shk}(z)) \times \mathsf{Exp}(\mathsf{shk}(z))$ where $w_1 \neq w_2$. Let $z_1 =$

$a_1\alpha^{i_1}u\beta^{j_1}b_1, z_2 = a_2\alpha^{i_2}u\beta^{j_2}b_2$ be distinct MAWs in $\mathcal{M}_4(w)$ where $\mathsf{shk}(z_1) = \mathsf{shk}(z_2) = w$. Assume towards a contradiction that z_1 and z_2 correspond to $(a'\alpha^{i'}u\beta^j b, a\alpha^i u\beta^{j'}b') \in \mathsf{Exp}(w) \times \mathsf{Exp}(w)$. This implies that, by Observation 3.1, $i = i_1 = i_2, j = j_1 = j_2, a = a_1 = a_2, b = b_1 = b_2$. Thus $z_1 = z_2$ holds, a contradiction. Hence, for any distinct MAWs $z_1, z_2 \in \mathcal{M}_4(w)$, z_1 and z_2 correspond to distinct elements of $\mathsf{Exp}(\mathsf{shk}(z)) \times \mathsf{Exp}(\mathsf{shk}(z))$. Since the number of elements (w_1, w_2) in $\mathsf{Exp}(\mathsf{shk}(z)) \times \mathsf{Exp}(\mathsf{shk}(z))$ such that $w_1 \neq w_2$ is $|\mathsf{Exp}(w)|(|\mathsf{Exp}(w)| - 1)$, this lemma holds. □

Since each MAW z corresponds to an element $(w_1, w_2) \in \mathsf{Exp}(\mathsf{shk}(z)) \times \mathsf{Exp}(\mathsf{shk}(z))$ such that $w_1 \neq w_2$, it is enough for the bound to sum up all $|\mathsf{Exp}(w)|^2$ such that $|\mathsf{Exp}(w)| \geq 2$ holds. Let $\mathcal{W}$ be the set of bridges w such that $|\mathsf{Exp}(w)| \geq 2$ or $w \in B_2 \cup B_3$. Let $\mathcal{X} = \sum_{w\in\mathcal{W}} |\mathsf{Exp}(w)|$. For considering such $\mathsf{Exp}(w)$, we also define a subset $\mathsf{Exp}^t_+(w)$ of $\mathsf{Exp}^t(w)$ as follows: For any string (bridge) w and integer $t > 0$,

$$\mathsf{Exp}^t_+(w) = \{w' \mid w' \in \mathsf{Exp}^t(w), |\mathsf{Exp}(w')| \geq 2\}.$$

We also use $\mathsf{Exp}_+(w)$ to denote $\mathsf{Exp}^1_+(w)$. Figure 3.7 shows an illustration for $\mathsf{Exp}^i(w)$, $\mathsf{Exp}^i_+(w)$, $\mathcal{W}$, and $\mathcal{X}$. We give the following lemma that explains relations between $\mathsf{Exp}^i(w)$, $\mathsf{Exp}^i_+(w)$, and $\mathcal{X}$.

Lemma 3.25.

$$\mathcal{X} = \sum_{w\in B_2\cup B_3} \left(|\mathsf{Exp}(w)| + \sum_{i=1}^{\lfloor m/2\rfloor-1} \sum_{z\in\mathsf{Exp}^i_+(w)} |\mathsf{Exp}(z)| \right).$$

Proof. Let z_{even} be a bridge where $R(z_{\mathsf{even}}) = 2i + 2$ for some $i \geq 1$. Notice that $\mathsf{shk}(z_{\mathsf{even}}) = c_1c_2 \in B_2$ for some distinct characters c_1, c_2. By the definition of $\mathsf{Exp}^i_+(\cdot)$, if $|\mathsf{Exp}(z_{\mathsf{even}})| \geq 2$, then $z_{\mathsf{even}} \in \mathsf{Exp}^i_+(c_1c_2)$. Let z_{odd} be a bridge where $R(z_{\mathsf{odd}}) = 2i+3$ for some $i \geq 1$. Notice that $\mathsf{shk}(z_{\mathsf{odd}}) = c_1c_2^kc_3 \in B_3$ for some characters c_1, c_2, c_3 and an integer $k \geq 1$. By the definition of $\mathsf{Exp}^i_+(\cdot)$, if $|\mathsf{Exp}(z_{\mathsf{odd}})| \geq 2$, then $z_{\mathsf{odd}} \in \mathsf{Exp}^i_+(c_1c_2^kc_3)$. Therefore the statement holds. □

This implies that $|\mathcal{M}_4| \leq \sum_{w\in\mathcal{W}} |\mathsf{Exp}(w)|^2 \leq \mathcal{X}^2$. Thus, if $\mathcal{X} \in O(m)$, $|\mathcal{M}_4| \in O(m^2)$.

We can also observe that $\sum_{i=1}^{\lfloor m/2\rfloor-1} \sum_{z\in\mathsf{Exp}^i_+(w)} |\mathsf{Exp}(z)|$ is the sum of the number of children of black nodes (which have more than a single child) in the tree for w. The number of leaves of the tree is an upper bound for the sum. It is also clear that $|\mathsf{Exp}(w)|$ can be bounded by the number of leaves of the tree. Consequently, we obtain $|\mathcal{X}| \in O(m)$ as in Lemma 3.26.

Lemma 3.26. $|\mathcal{X}| \in O(m)$.

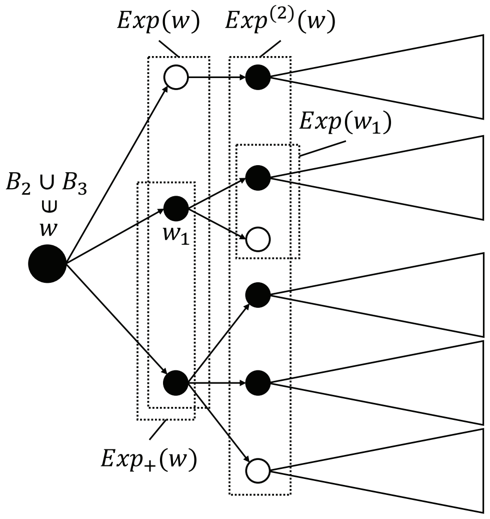


Figure 3.7: This tree shows an illustration for $\mathsf{Exp}^i(w), \mathsf{Exp}_+^i(w), \mathcal{W}$, and $\mathcal{X}$. The root node represents a bridge $w \in B_2 \cup B_3$. The set of children of the root corresponds to $\mathsf{Exp}(w)$, namely, each child x represents a bridge such that $\mathsf{shk}(x) = w$. Each black node represents a bridge x such that $|\mathsf{Exp}(x)| \geq 2$ (i.e., each black node has at least two children) or the root. Let $W(w)$ be the set of nodes consisting of all the black nodes in the tree rooted at a bridge $w \in B_2 \cup B_3$. Then $\mathcal{W}$ is the union of $W(w)$ for all $w \in B_2 \cup B_3$, and $\mathcal{X}$ is the total number of children of black nodes in $\mathcal{W}$.

Proof. By Lemma 3.25 and the above discussion, we have

$$\begin{aligned} \mathcal{X} &= \sum_{w \in B_2 \cup B_3} \left(|\mathsf{Exp}(w)| + \sum_{i=1}^{\lfloor m/2 \rfloor - 1} \sum_{z \in \mathsf{Exp}_+^i(w)} |\mathsf{Exp}(z)| \right) \\ &\leq \sum_{w \in B_2 \cup B_3} 2\#w \\ &\leq 2\left((m-1) + (m-2)\right) \in O(m). \end{aligned}$$

□

We are ready to prove Lemma 3.23:

Proof of Lemma 3.23. $|\mathcal{M}_4| \leq \sum_{w \in \mathcal{W}} |\mathsf{Exp}(w)|^2 \leq |\mathcal{X}|^2 \leq (2(2m-3))^2 \in O(m^2)$. □

3.2.4 Efficient representations of MAWs for RLE strings

Consider a string T that contains σ' distinct characters. In this section, we present compact data structures that can output every MAW for T upon query, using a total of $O(m)$ space, where $m = R(T)$ is the size of $\mathsf{rle}(T)$. We will prove the following theorem:

Theorem 3.5. *There exists a data structure* D *of size* $O(m)$ *which can output all MAWs for string* T *in* $O(|\mathsf{MAW}(T)|)$ *time, where* m *is the RLE-size of* T*.*

In our representation of MAWs that follows, we store $\mathsf{rle}(T)$ explicitly with $O(m)$ space. The following is a general lemma that we can use when we output a MAW from our data structures.

Lemma 3.27. *For each MAW* $w \in \mathsf{MAW}(T)$*,* $\mathsf{rle}(w)$ *of size* $R(w)$ *can be retrieved in* $O(R(w))$ *time from a tuple* (a, i, s, t, b, j) *and* $\mathsf{rle}(T)$*, where* $a, b \in \Sigma$*,* $0 \leq i, j \leq |T|$*, and* $0 \leq s, t \leq m$*.*

Proof. When $R(w) = 1$ (i.e. $w \in \mathcal{M}_1$), then since w is of the form a^i with $i \geq 1$, we can simply represent it by $(a, i, 0, 0, 0, 0)$.

When $R(w) \geq 2$, then let $w = aub$. When $aub \in \mathcal{M}_2$, then $w = ab$ and thus it can be simply represented by $(a, 1, 0, 0, b, 1)$. When $aub \in \mathcal{M}_3 \cup \mathcal{M}_4$, then $a \neq u[1]$ and $b \neq u[|u|]$. Hence it can be represented by $(a, 1, s, t, b, 1)$ where $r_s \cdots r_t = \mathsf{rle}(u)$. When $aub \in \mathcal{M}_5$, then $a = u[1]$ or $u[|u|] = b$. Let i, j be the maximal integers such that $a^i u' b^j = aub$. We can represent it by (a, i, s, t, b, j) with $r_s \cdots r_t = \mathsf{rle}(u')$. □

For ease of discussion, in what follows, we will identify each MAW w with its corresponding tuple (a, i, s, t, b, j) which takes $O(1)$ space.

Representation for $\mathcal{M}_1$

We have shown that $|\mathcal{M}_1| = \sigma$ (Lemma 3.19), however, σ can be larger than σ' and m. Still, a simple representation for $\mathcal{M}_1$ exists, as follows:

Lemma 3.28. *There exists a data structure* D_1 *of* $O(\sigma') \subseteq O(m)$ *space that can output each MAW in* $\mathcal{M}_1$ *in* $O(1)$ *time.*

Proof. For ease of explanation, assume that the string T is over the integer alphabet $\Sigma = \{1, \ldots, \sigma\}$ and let $\Sigma' = \{c_1, \ldots, c_{\sigma'}\} \subseteq \{1, \ldots, \sigma\}$. Let $M = \langle c_1^{p_1}, \ldots, c_{\sigma'}^{p_{\sigma'}} \rangle$ be the list of Type-1 MAWs in $\mathcal{M}_1$ that are runs of characters in Σ', sorted in the lexicographical order of the characters, i.e. $1 \leq c_1 < \cdots < c_{\sigma'} \leq \sigma$. We store M explicitly in $O(\sigma')$ space. When we output

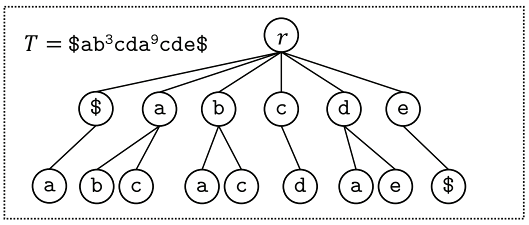


Figure 3.8: The trie D_2 for string $T = \$\mathtt{ab}^3\mathtt{cda}^9\mathtt{cde}\$$. A bigram ab with $a \neq b$, $a, b \in \Sigma'$ is in $\mathcal{M}_2$ iff ab is not in this trie D_2. For instance, ae and db are MAWs of T.

each MAW in $\mathcal{M}_1$, we test the numbers (i.e. characters) in $\Sigma = \{1, \ldots, \sigma\}$ incrementally, and scan M in parallel: For each $c = 1, \ldots, \sigma$ in increasing order, if $c^p \in M$ with some $p > 1$ then we output c^p, and otherwise we output c. □

Representation for $\mathcal{M}_2$

Recall that $|\mathcal{M}_2| \in O(\sigma'^2) \subseteq O(m^2)$ and this bound is tight in the worst case. Therefore we cannot store all elements of $\mathcal{M}_2$ explicitly, as our goal is an $O(m)$-space representation of MAWs. Nevertheless, the following lemma holds:

Lemma 3.29. *There exists a data structure* D_2 *of* $O(m)$ *space that can output each MAW in* $\mathcal{M}_2$ *in* $O(1)$ *amortized time.*

Proof. If $|\mathcal{M}_2| \in O(m)$, then we explicitly store all elements of $\mathcal{M}_2$.

If $|\mathcal{M}_2| \in \Omega(m)$, then let D_2 be the trie that represents all bigrams that occur in T. See Figure 3.8 for a concrete example of D_2. Note that for any pair $a, b \in \Sigma'$ of *distinct* characters both occurring in T, ab is either in D_2 or in $\mathcal{M}_2$. Since the number of such pairs a, b is $\sigma'(\sigma'-1)$, we have that $\sigma'^2 = \Theta(|\mathsf{D}_2| + |\mathcal{M}_2|)$, where $|\mathsf{D}_2|$ denotes the size of the trie D_2. Since $|\mathsf{D}_2| < m$, we have $\sigma'^2 \in O(|\mathcal{M}_2| + m)$. Suppose that the character labels of the out-going edges of each node in D_2 are lexicographically sorted. When we output each element in $\mathcal{M}_2$, we test every bigram ab such that $a \neq b$ and $a, b \in \Sigma'$ in the lexicographical order, and traverse D_2 in parallel in a depth-first manner. We output ab if it is not in the trie D_2. This takes $O(\sigma'^2 + |\mathsf{D}_2|) \subseteq O(|\mathcal{M}_2| + m) \subseteq O(|\mathcal{M}_2|)$ time, since $|\mathcal{M}_2| \in \Omega(m)$. □

Representation for $\mathcal{M}_3$

Recall that the number of MAWs of Type-3 in $\mathcal{M}_3$ is unbounded by the RLE size m (Lemma 3.21). Nevertheless, we show that there exists a compact $O(m)$-space data structure that can report each MAW in $\mathcal{M}_3$ in $O(1)$ time.

Notice that, by definition, a MAW aub of Type-3 is a bridge and therefore, it is of the form ac^kb with $c \in \Sigma'_T \setminus \{a, b\}$ and $k \geq 1$. Furthermore, ac^k and c^kb appear in T. Then, the following observation gives an idea of representation.

Observation 3.2. *A bridge ac^kb is in $\mathcal{M}_3$ iff (i) $ac^kb \notin \mathcal{BS}_c(T)$ and (ii) $ac^{k'} \in \mathcal{L}_c$ with $k' \geq k$ and (iii) $c^{k'}b \in \mathcal{R}_c$ with $k' \geq k$.*

Let $\mathcal{BS}_c(T)$ be the ordered set of bridge substrings z of T with $R(z) = 3$ whose middle run consists of c. For each character $a \in \Sigma'_T \setminus \{c\}$, let i be the largest exponent such that ac^i occurs in T. Let $\mathcal{L}_c$ be the set of such ac^i's for all characters $a \in \Sigma'_T \setminus \{c\}$. Similarly, let $\mathcal{R}_c$ be the set of c^jb's for all characters $b \in \Sigma'_T \setminus \{c\}$, where j is the largest exponent such that cb^j occurs in T. Each element ac^kb of $\mathcal{BS}_c(T)$ is sorted in a's lexicographical order. If a is the same, it is sorted in b's lexicographical order. If b is the same too, it is sorted in ascending order of the exponent k. Each element ac^k of $\mathcal{L}_c$ is sorted in a's lexicographical order. Each element c^kb of $\mathcal{R}_c$ is sorted in b's lexicographical order.

See Figure 3.9 for a concrete example for $\mathcal{L}_c$ and $\mathcal{R}_c$.

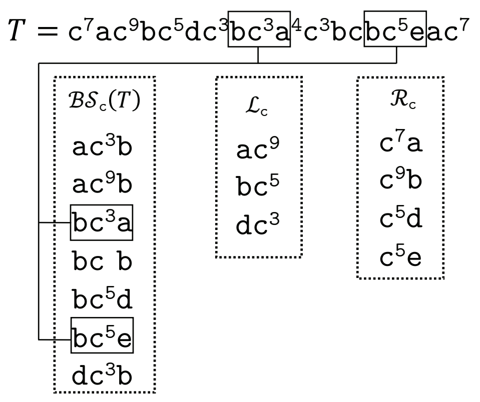


Figure 3.9: $\mathcal{BS}_\mathtt{c}$, $\mathcal{L}_\mathtt{c}$, and $\mathcal{R}_\mathtt{c}$ for string $T = \mathtt{c^7ac^9bc^5dc^3bc^3a^4c^3bcbc^5eac^7}$ and character $\mathtt{c}$.

By the observation 3.2, we have the following lemma:

Lemma 3.30. *There exists a data structure D_3 of $O(m)$ space that can output all MAWs in $\mathcal{M}_3$ in $O(|\mathcal{M}_3|)$ time.*

Proof. Given $\mathcal{BS}_c, \mathcal{L}_c$ and $\mathcal{R}_c$, we can enumerate all Type-3 MAWs. For each pair ac^i and c^jb of $\mathcal{L}_c$ and $\mathcal{R}_c$, $acb, ac^2b, \ldots, ac^{i'}b$ are the candidates of MAWs of T where $i' = \min(i, j)$. Note that $ac^{i'+1}b, ac^{i'+2}b, \ldots$ are not MAWs of T. For each candidate ac^kb, it is a MAW of T if $ac^kb \notin \mathcal{BS}_c(T)$. Let $\sigma_{x,y}$ denote the pair of the x-th element of $\mathcal{L}_c$ and the the y-th element of $\mathcal{R}_c$. See the algorithm illustrated in Algorithm 1. The algorithm first scans $\sigma_{1,1}$ to $\sigma_{1,|\mathcal{R}_c|}$, and then $\sigma_{2,1}$ to $\sigma_{2,|\mathcal{R}_c|}$, and so forth, and finally scans $\sigma_{|\mathcal{L}_c|,1}$ to $\sigma_{|\mathcal{L}_c|,|\mathcal{R}_c|}$. Then, $\mathcal{BS}_c(T)$ has been sorted in the same order as the pairs the algorithm scans. In other words, the algorithm can verify whether the candidate is a MAW of T by comparing it with the current first element of $\mathcal{BS}_c(T)$ that the algorithm has not verified yet. Therefore, one can determine whether the candidate ac^kb is a MAW or not in $O(1)$ time each.

The correctness of the algorithm follows from Observation 3.2. Since $\sum_{c\in\Sigma'_T} |\mathcal{BS}_c(T)| \in O(m)$, the total space requirement of the data structure for all characters in Σ'_T is $O(m)$.

Let us consider the time complexity. Let occ_p be the MAW of $\mathcal{M}_3$, and occ_n be the The algorithm takes $O(|\mathcal{M}_3| + |\mathcal{BS}|) \subseteq O(m + |\mathcal{M}_3|)$ time, where $\mathcal{BS} = \bigcup_{c\in\Sigma'_T} \mathcal{BS}_c$. When $|\mathcal{M}_3| \in O(m)$, we can represent and store all elements in $\mathcal{M}_3$ in a total of $O(m)$ space by using Lemma 3.27. Therefore, the algorithm takes a total of $O(|\mathcal{M}_3|)$ time. □

Algorithm 1 Enumerate all Type-3 MAWs

```
Input: $\mathcal{BS}_c$, $\mathcal{L}_c$ and $\mathcal{R}_c$
Output: all Type-3 MAW is output
 1: for each $ac^i$ in $\mathcal{L}_c$ do
 2:   for each $c^jb$ in $\mathcal{R}_c$ do
 3:     $i' = \min(i, j)$
 4:     for $k$ with 1 to $i'$ do
 5:       if $ab^kc \in \mathcal{BS}_c(T)$ then
 6:         output $ac^kb$
 7:       end if
 8:     end for
 9:   end for
10: end for
```

Representation for $\mathcal{M}_4$

Recall that $|\mathcal{M}_4| \in O(m^2)$ and this bound is tight in the worst case. Therefore we cannot store all elements of $\mathcal{M}_4$ explicitly, as our goal is an $O(m)$-space representation of MAWs. Nevertheless, the following lemma holds:

Lemma 3.31. *There exists a data structure D_4 of $O(m)$ space that can output each MAW in $\mathcal{M}_4$ in $O(1)$ amortized time.*

Our data structure D_4 is based on the discussion in Section 3.2.3. We consider the following bipartite graph $G_w = (V_L \cup V_R, E)$ for any bridge $w \in \mathcal{W}$. We can identify each bridge $a\alpha^i u\beta^j b \in \mathsf{Exp}(w)$ by representing the bridge as a 4-tuple (a, i, j, b). Let F_w be the set of 4-tuples which represents all elements in $\mathsf{Exp}(w)$. Two disjoint sets V_L, V_R of vertices and set E of edges are defined as follows:

$$\begin{aligned} V_L &= \{(a, i) \mid \exists(a, i, j, b) \in F_w\}, \\ V_R &= \{(j, b) \mid \exists(a, i, j, b) \in F_w\}, \\ E &= \{((a, i), (j, b)) \mid \exists(a, i, j, b) \in F_w\}. \end{aligned}$$

V_L (resp. V_R) represents the set of the left (resp. right) parts of bridges in $\mathcal{W}$. For each edge in E represents a bridge in $\mathcal{W}$. This implies that $|E| = |\mathsf{Exp}(w)|$. Assume that all vertices in V_L (resp. V_R) are sorted in non-decreasing order w.r.t. the value i (resp. j) which represents the exponent of corresponding run. For any $k \in [1, |V_L|]$ and $k' \in [1, |V_R|]$, $v_L(k) = (\mathsf{c}_L(k), \mathsf{e}_L(k))$ denotes the k-th vertex in V_L, and $v_R(k') = (\mathsf{c}_R(k'), \mathsf{e}_R(k'))$ denotes the k'-th vertex in V_R. For any vertex $v_L(k) \in V_L$ and $v_R(k') \in V_R$, we also define

$$E^{LR}_{max}(k) = \max\{\mathsf{e}_R(i) \mid \exists(v_L(k), v_R(i)) \in E\},$$

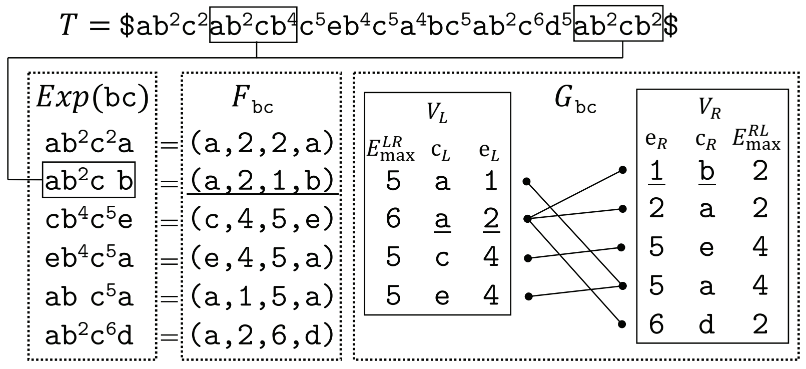
$$E^{RL}_{max}(k') = \max\{\mathsf{e}_L(i) \mid \exists(v_R(i), v_R(k)) \in E\}.$$

Figure 3.10 gives an illustration for this graph. Due to Observation 3.1, each MAW z of Type-4 corresponds to an element of $\mathsf{Exp}(w) \times \mathsf{Exp}(w)$ where $z^{(1)} = w$. By this idea, we detect each MAW as a pair of vertices in $V_L \times V_R$ which is not an edge in E. The following lemma explains all MAWs which can be represented by the graph.

Lemma 3.32. *For any vertices $v_L(k) \in V_L$ and $v_R(k') \in V_R$ of $G_{\alpha u\beta}$, the string $\mathsf{c}_L(k)\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}\mathsf{c}_R(k')$ is a MAW iff the following three conditions hold (see also Figure 3.11 for an illustration):*

- $(v_L(k), v_R(k')) \notin E$,
- $E^{LR}_{max}(k) \geq \mathsf{e}_R(k')$, *and*
- $E^{RL}_{max}(k') \geq \mathsf{e}_L(k)$.

Proof. If $(v_L(k), v_R(k')) \notin E$, $\mathsf{c}_L(k)\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}\mathsf{c}_R(k')$ is an absent word. $E^{LR}_{max}(k) \geq \mathsf{e}_R(k')$ and $E^{RL}_{max}(k') \geq \mathsf{e}_L(k)$ implies that $\mathsf{c}_L(k)\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}$ and $\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}\mathsf{c}_R(k')$ occur in the string. Thus $\mathsf{c}_L(k)\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}\mathsf{c}_R(k')$ is a MAW.



$T = \$ab^2c^2\boxed{ab^2cb^4}c^5eb^4c^5a^4bc^5ab^2c^6d^5\boxed{ab^2cb^2}\$$
$Exp(bc)$
F_{bc}
G_{bc}
V_L
V_R

$Exp(\mathtt{bc})$		$F_{\mathtt{bc}}$
ab^2c^2a	=	(a,2,2,a)
$\boxed{ab^2c\ b}$	=	(a,2,1,b)
cb^4c^5e	=	(c,4,5,e)
eb^4c^5a	=	(e,4,5,a)
$ab\ c^5a$	=	(a,1,5,a)
ab^2c^6d	=	(a,2,6,d)

$E^{LR}_{\max}$	c_L	e_L
5	a	1
6	a	2
5	c	4
5	e	4

e_R	c_R	$E^{RL}_{\max}$
1	b	2
2	a	2
5	e	4
5	a	4
6	d	2

Figure 3.10: This figure shows $G_{\mathtt{bc}}$ for $T = \$\mathtt{ab^2c^2ab^2cb^4c^5eb^4c^5a^4bc^5ab^2c^6d^5ab^2cb^2}\$$. For a bridge bc, Exp(bc) has 6 bridges. $F_{\mathtt{bc}}$ contains 6 tuples which represents all bridges in Exp(bc). For instance, a bridge $\mathtt{ab^2cb} = (\mathtt{a}, 2, 1, \mathtt{b})$ where the first character is a, the exponent of the second run is 2, the exponent of the second last run is 1, and the last character is b. V_L is the set of pairs by the left-half of elements in $F_{\mathtt{bc}}$. In this example, V_L has 4 vertices $\{(\mathtt{a}, 1), (\mathtt{a}, 2), (\mathtt{c}, 4), (\mathtt{e}, 4)\}$ which are sorted in non-decreasing order of the second key (representing its exponent). V_R is the symmetric set for the right parts. Each bridge corresponds to an edge. For example, the second bridge $\mathtt{ab^2cb}$ in the figure corresponds to the edge from the second vertex $(\mathtt{a}, 2)$ in V_L to the first vertex $(1, \mathtt{b})$ in V_R. Since the number of bridges in Exp(bc)($F_{\mathtt{bc}}$) is 6, the graph has 6 edges.

On the other hand, if $(v_L(k), v_R(k')) \in E$, $\mathtt{c}_L(k)\alpha^{\mathtt{e}_L(k)}u\beta^{\mathtt{e}_R(k')}\mathtt{c}_R(k')$ occurs in the text. $E^{LR}_{max}(k) < \mathtt{e}_R(k')$ implies that $\mathtt{c}_L(k)\alpha^{\mathtt{e}_L(k)}u\beta^{\mathtt{e}_R(k')}$ does not occur in the string. $E^{RL}_{max}(k') < \mathtt{e}_L(k)$ implies that $\alpha^{\mathtt{e}_L(k)}u\beta^{\mathtt{e}_R(k')}\mathtt{c}_R(k'))$ does not occur in the string. Thus all three conditions hold if $\mathtt{c}_L(k)\alpha^{\mathtt{e}_L(k)}u\beta^{\mathtt{e}_R(k')}\mathtt{c}_R(k')$ is a MAW. □

Proof of Lemma 3.31. Let x be the number of outputs. If $x < m$, we can just store all the MAWs themselves. Assume that $x \in \Omega(m)$.

For all bridge $w = \alpha u\beta \in \mathcal{W}$, G_w represents all MAWs which correspond to elements in $\mathsf{Exp}(w) \times \mathsf{Exp}(w)$. Our data structure D_4 consists of G_w for any $w \in \mathcal{W}$. It is clear that G_w can be stored in $O(|\mathsf{Exp}(w)|)$ space. This implies that the size of D_4 is linear in $\mathcal{X}$, namely, D_4 can be stored in $O(m)$ space (Lemma 3.26).

We can output all MAWs which are represented by G_w based on Lemma 3.32 (see Algorithm 2). For the k-th vertex $v_L(k)$, C represents all vertices $v_R(k')$ in V_B such that $(v_L(k), v_R(k')) \notin E$ and $E^{RL}_{max}(k') \geq \mathtt{e}_L(k)$ (the first and third condition in Lemma 3.32). For each vertex in C, if $E^{LR}_{max}(k) \geq \mathtt{e}_R(k')$ (the second condition in Lemma 3.32), the algorithm outputs a MAW

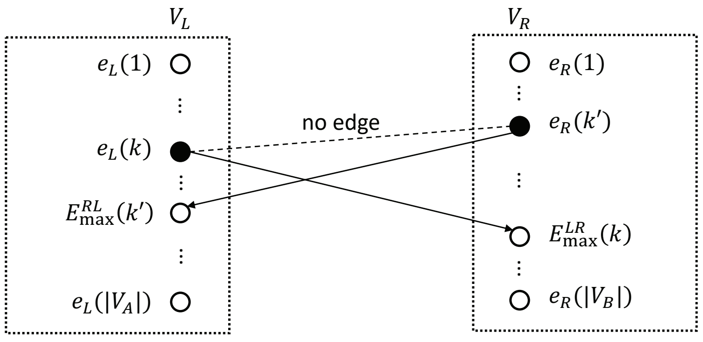


Figure 3.11: This is an illustration for Lemma 3.32. For the k-th vertex $v_L(k) \in V_L$ and k'-th vertex $v_R(k') \in V_R$, this graph satisfies the three conditions of the lemma.

Algorithm 2 Compute all MAWs in $\mathcal{M}_4$

Input: bipartite graph $G_{\alpha u \beta} = (V_L, V_R, E)$

Output: all MAWs in $\mathcal{M}_4$ that are associated by $\alpha u \beta$, $a\alpha^{k_1}u\beta^{k_2}b$ for $a, b \in \Sigma, k_1, k_2 \in \mathbb{N}$

```
 1: $C_R \leftarrow V_R$
 2: for each $v_L(k) \in V_L$ do
 3:     $C = \{v_R(k') \in C_R \mid \mathsf{e}_R(k') \leq E^{LR}_{max}(k)\} \setminus \{v \mid (v_L(k), v) \in E\}$
 4:     for each $v_R(k') \in C$ do
 5:         if $E^{RL}_{max}(k') \geq \mathsf{e}_L(v_L(k))$ then
 6:             output $\mathsf{c}_L(k)\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}\mathsf{c}_R(k')$
 7:         else
 8:             $C_R \leftarrow C_R \setminus \{v_R(k')\}$
 9:         end if
10:     end for
11: end for
```

$\mathsf{c}_L(k)\alpha^{\mathsf{e}_L(k)}u\beta^{\mathsf{e}_R(k')}\mathsf{c}_R(k')$. Then the running time of our algorithm is $O(x + \sum_{w \in \mathcal{W}} |G_w|) \subseteq O(x + m) = O(x)$, since $x \in \Omega(m)$. □

Representation for $\mathcal{M}_5$

Lemma 3.33. *There exists a data structure of size $O(m)$ that outputs each element of $\mathcal{M}_5$ in $O(1)$ time.*

Proof. By Lemma 3.22, $|\mathcal{M}_5| \in O(m)$. Recall that an element of M_5 can be as long as $O(n)$. However, using Lemma 3.27 we can represent and store all elements in $\mathcal{M}_5$ in a total of $O(m)$ space. It is trivial that each stored element can be output in $O(1)$ time. $\square$

3.2.5 Constructing representations of MAWs for RLE strings

We have shown our representations which can efficiently output MAWs for each type. In this section, we how to construct these representations in $O(m \log m)$ total time, for a given RLE string of size m consisting of σ' distinct characters.

Construction of D_1

Lemma 3.34. *We can construct D_1 in $O(m \log \sigma')$ time.*

Proof. The goal of this lemma is how to construct D_1 that we explained in Lemma 3.28. For each ith run, we scan ${a_i}^{p_i}$ and store the maximum exponent for each distinct character. Note that we can omit the characters that do not occur in T, and therefore we can compute D_1 in $O(m \log \sigma')$ time. $\square$

Construction of D_2

Lemma 3.35. *We can construct D_2 in $O(m \log \sigma')$ time.*

Proof. For each pair of adjacent ith and $(i+1)$th runs, we scan $a_i a_{i+1}$ to store all bigrams that occur in T into a trie.

During our construction of the trie, we increment a counter e each time a new bigram is inserted into a trie. If the final number e of edges in the resulting trie is at most $\sigma(\sigma-1)-m$, then we store this trie as D_2. We can compute this trie in $O(m \log \sigma')$ time.

As soon as the counter value e exceeds $\sigma(\sigma-1)-m$ at some point, then we can recognize that there are at most m Type-2 MAWs of T. In this case, we rearrange the data structure D'_2 that explicitly stores all Type-2 MAWs in T. We can compute D'_2 using $O(m)$ additional time, using the representation strategy for $\mathcal{M}_2$. $\square$

Construction of D_3

Lemma 3.36. *We can construct D_3 in $O(m \log m)$ time.*

Proof. D_3 consists of $\mathcal{BS}_b$, $\mathcal{L}_b$ and $\mathcal{R}_b$. We first compute $\mathcal{BS}_b$, which is the set of all bridge of size 3 occurring T such that its middle run has b. For each bridge of size 3, we classify them based on the character of its middle run b. In this way we first compute $\mathcal{BS}_b$ for each $b \in \Sigma_T$ in $O(m)$ time. We can sort each $\mathcal{BS}_b$ by the exponent of its middle character in $O(|\mathcal{BS}_b| \log |\mathcal{BS}_b|)$ time. The sum of $|\mathcal{BS}_b|$ for any $b \in \Sigma_T$ is at most $m-2$ since their elements are bridges of size

3 that occurs in T. Therefore, we can sort all $\mathcal{BS}_b$ in $O(m \log m)$ time in total. $\mathcal{L}_b$ and $\mathcal{R}_b$ is computed in $O(|\mathcal{BS}_b| \log |\mathcal{BS}_b|)$ time from each $\mathcal{BS}_b$, which results in a total D_3 computation time of $O(m \log m)$. $\square$

Construction of D_4

If we know the set $\mathsf{Exp}(\alpha u \beta)$, we can construct each bipartite graph $G_{\alpha u \beta} = (V_L \cup V_R, E)$ by sorting in the exponents order of α and β. We can also compute all G from all Exp that we need for Type-4 MAW in $O(m \log m)$ time by Lemma 3.26. Then, our target is how to output all Exp for Type-4 MAWs.

Any element of $\mathsf{Exp}(\alpha u \beta)$ has $\alpha u \beta$ as its substring. Moreover, the substring always starts at the last position of the second run. Therefore, as we can see in Figure 3.12, there is a single range in the sorted tRLE suffix array that contains all such substrings. We call this range a block below. Let t be the first position of the block, and t' be the last position of the block in the tRLESA. Let k be the the RLE size of u, in other words $u = u_1 \dots u_k$. Then, the three following statements hold:

1. for any i with $t < i \le t'$, $\mathsf{tRLELCP}(i) \ge k + 2$.
2. $\mathsf{tRLELCP}(t) < k + 2$.
3. $\mathsf{tRLELCP}(t' + 1) < k + 2$.

Using these conditions, we can compute the all $\mathsf{Exp}(\alpha u \beta)$ that we need for D_4. Let $\mathsf{Br}(i, j)$ be the prefix bridge of $\mathsf{tRLE}(i)$ such that $R(\mathsf{Br}(i, j)) = j$. Algorithm 3 shows a naïve method which computes all $\mathsf{Exp}(\alpha u \beta)$ that have common RLE size of u.

However, the naive method of Algorithm 3 is not enough to compute all in desired Exp in $O(m \log m)$ time. The issues are the following:

1. The algorithm may multiply count the same string. It happens when $\mathsf{Br}(x - 1, k + 2)$ is already in $\mathsf{Exp}(\mathsf{Br}(x, k))$.
2. The algorithm may create Exp such that $|\mathsf{Exp}(\alpha u \beta)| = 1$. It is never used for D_4.
3. Due to the two above issues, it can take $O(m^2)$ time to compute all $\mathsf{Exp}(\alpha u \beta)$ for D_4 since there can exist useless Exp and thus meaningless counting operations can be performed in Algorithm 3.

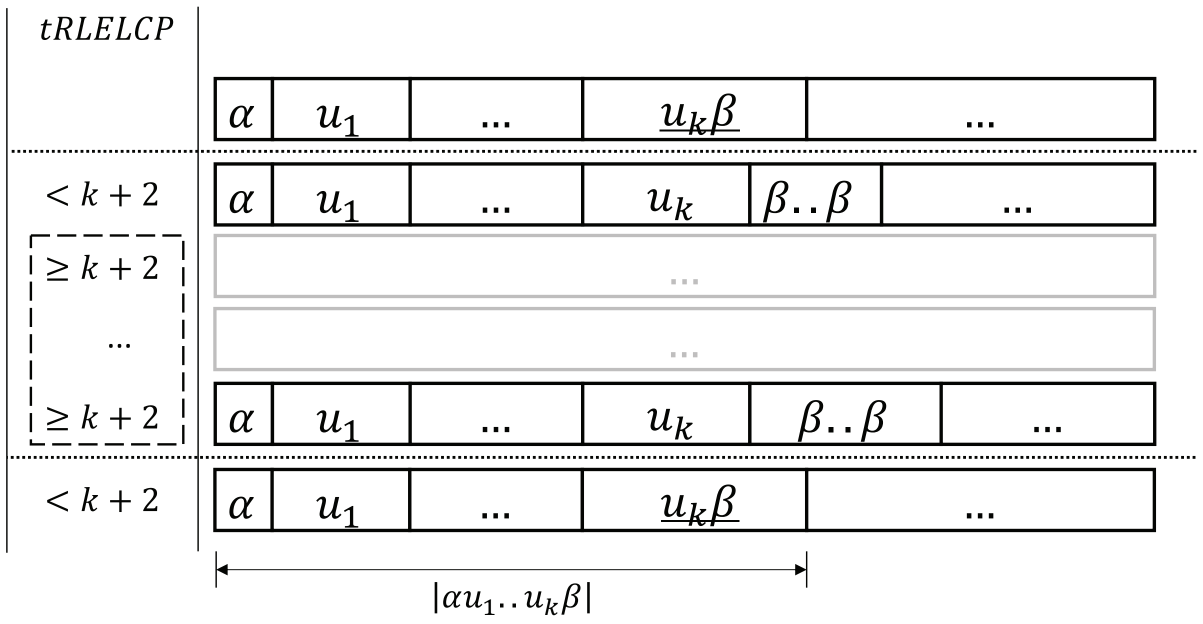


Figure 3.12: All occurrence of $\alpha u\beta = \alpha u_1 \dots u_k\beta$ in T is laid out in a line inside lexicographically ordered tRLE suffix list. We obtain $\mathsf{Exp}(\alpha u\beta)$ from a corresponding maximum block that tRLELCP is greater than $R(\alpha u\beta)$. For each i that belongs to the block, ($\mathsf{tRLESA}[i]-1$)-th tRLE suffix is included in $\mathsf{Exp}(\alpha u\beta)$.

To resolve the first issue, we introduce SP which stores the nearest position of tRLESA that occurs the string s before the tRLE suffix. We show an example of SP in Figure 3.13. In this case, Algorithm 3 tries to count $\alpha_2\alpha..\alpha u_1...u_k\beta..\beta\beta_2$ twice while scanning i. However, after $SP(s)$ returns i', the modified algorithm verifies that $\mathsf{tRLESA}[i'] - 1$-th tRLE suffix has $\alpha_2\alpha..\alpha$ as its prefix. Then, if $LCP(\mathsf{tRLESA}[i'], \mathsf{tRLESA}[i]) > k+2$, we can see that $\alpha_2\alpha..\alpha u_1...u_k\beta..\beta\beta_2$ occurs twice or more. We can easily maintain SP by placing $SP(\alpha_2\alpha^p) = i'$ whenever we encounter $\alpha_2\alpha^p$ while computing i'-th elements of tRLESA. Then, SP is always updated by largest i'.

To resolve the second issue, we observe that we just need to check $\mathsf{tRLELCP}[i-1]$ to confirm whether the block has strictly two or more elements, or not.

To resolve the third issue, we refine the algorithm so that it can process different lengths k in parallel. The two ideas we mentioned above use the common SP, and verification of $\mathsf{tRLELCP}[i-1]$ is also a common operation. By Lemma 3.26, in the condition that there are no useless $\mathsf{Exp}(\alpha u\beta)$ (in other words $|\mathsf{Exp}(\alpha u\beta)| = 1$) and no duplicated counting operations, there are $O(m)$ elements that the algorithm counts.

Lemma 3.37. *We can construct* D_4 *in* $O(m \log m)$ *time.*

Algorithm 3 Compute all $\mathsf{Exp}(\alpha u \beta)$ for D_4 where k is the RLE size of u

Input: $\mathsf{tRLESA}, \mathsf{tRLELCP}$, initialized trie SP

Output: All $\mathsf{Exp}(\alpha u \beta)$ with $|\mathsf{Exp}(\alpha u \beta)| \geq 2$ and the RLE size of u is k.

```
1: for each $i$ from 1 to $m$ do
2:     $x = \mathsf{tRLESA}[i]$
3:     if $\mathsf{tRLELCP}[i] \geq k$ then
4:         add $\mathsf{Br}(x-1, k+2)$ to $\mathsf{Exp}(\mathsf{Br}(x, k))$
5:     end if
6: end for
```

Proof. Algorithm 4 computes all Exp that are required for computing D_4. For any $\mathsf{tRLESA}[t]$ with $i' < t < i$, $\mathsf{tRLE}(t)$ has the prefix $\mathsf{Br}(x, \mathsf{LCP}(\mathsf{tRLE}(x), \mathsf{tRLE}(x')))$.

If $j < \mathsf{LCP}(\mathsf{tRLE}(x), \mathsf{tRLE}(x')))$, we do not have to include $\mathsf{Br}(x-1, j+2)$ in $\mathsf{Br}(x, j)$ since $\mathsf{Br}(x-1, j+2) = \mathsf{Br}(x'-1, j+2)$ is included in $\mathsf{Br}(x, j)$. If $j \geq \mathsf{LCP}(\mathsf{tRLE}(x), \mathsf{tRLE}(x')))$, $\mathsf{Br}(x-1, j+2)$ has not been included in $\mathsf{Br}(x, j)$ yet since $\mathsf{Br}(x-1, j+2) \neq \mathsf{Br}(x'-1, j+2)$.

Therefore, the total number of element added to Exp in this algorithm is exactly equal to the number of elements of all Exp, which is $O(m)$ by Lemma 3.26.

Accessing any position of SP takes $O(\log m)$ time, $\mathsf{LCP}(\mathsf{tRLE}(x), \mathsf{tRLE}(x'))$ is $O(1)$ time by range minimum queries of $[x', x]$. Considering there are m steps, Algorithm 4 runs in $O(m \log m)$ time in total. □

Construction of D_5

In Section 3.2.4 we showed that D_5 naïvely stores all Type-5 MAWs. Therefore, we prove that we can enumerate all MAWs in $\mathcal{M}_5$, which is enough to prove that we can construct D_5 in its time. We divide each $u = u_1 u_2 \cdot u_{m'}$ in $\mathcal{M}_5$ into 5 subtypes.

1. $m' = 2$ and $|u_1| = 1$. (Type 5-1)
2. $m' = 2$ and $|u_{m'}| = 1$. (Type 5-2)
3. $m' \geq 3$ and $|u_1| = 1$. (Type 5-3)
4. $m' \geq 3$ and $|u_{m'}| = 1$. (Type 5-4)
5. (else) $|u_1| \geq 2, |u_{m'} \geq 2|$. (Type 5-5)

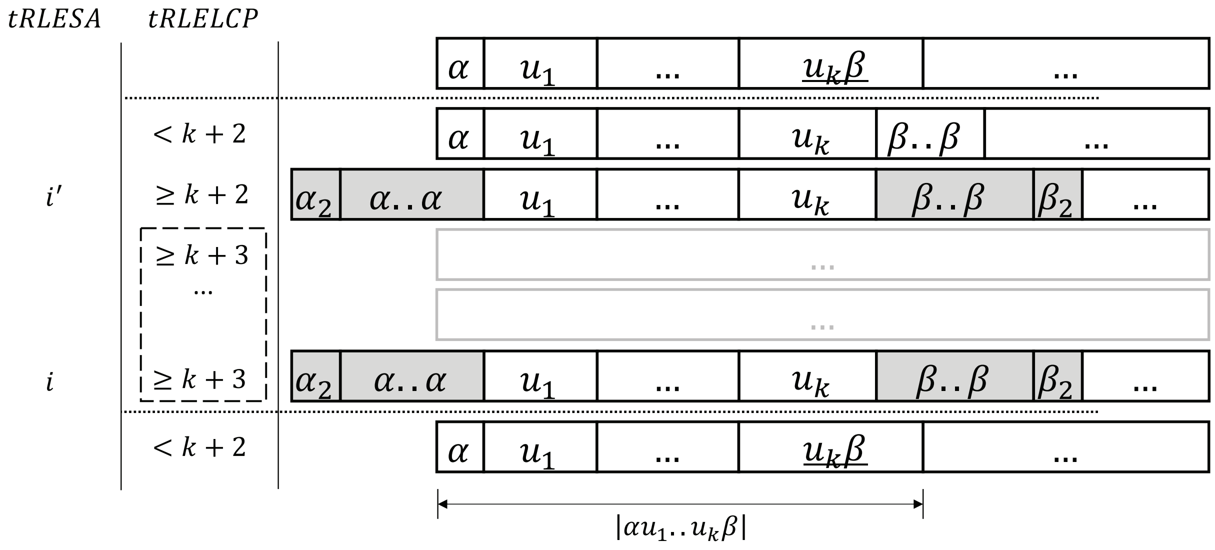


Figure 3.13: $(\mathsf{tRLESA}[i]-1)$-th tRLE suffix starts with $\alpha_2\alpha^p$. i' is the maximum number such that $i > i'$ and $(\mathsf{tRLESA}[i']-1)$-th tRLE suffix starts with $\alpha_2\alpha^p$. At the time computing i-th element of tRLESA, $SP(\alpha_2\alpha^p)$ returns such i'. Assume that the longest common prefix of $(\mathsf{tRLESA}[i])$-th tRLE suffix and $(\mathsf{tRLESA}[i'])$-th tRLE suffix is greater than $k+2$. Then, there are two same strings $\alpha_2\alpha..\alpha u\beta..\beta\beta_2$, which should be included in $\mathsf{Exp}(\alpha_2\alpha..\alpha u\beta..\beta\beta_2)$ just once.

α, β, γ are characters such that $\alpha \neq \beta, \beta \neq \gamma, \gamma \neq \alpha$. We can represent Type 5-1 MAW in the form of $u = \alpha\beta^k$ with $k \geq 2$. Then the following lemma holds.

Lemma 3.38. *Proposition A equals to B.*

- *A:* $\beta^{k'} \in \mathsf{D}_1$ *with* $k' \geq k$ *and* $\alpha\beta^{k-1} \in \mathcal{L}_\beta$ *in* D_3
- *B:* $\alpha\beta^k$ *is MAW of* T.

Proof. If $\beta^{k'} \in \mathsf{D}_1$ with $k' \geq k$, β^k occurs in T. If $\alpha\beta^{k-1} \in \mathcal{L}_\beta$, $\alpha\beta^{k-1}$ occurs in T, and $\alpha\beta^k$ does not occurs in T. Therefore, $A \Rightarrow B$ is true. If $\alpha\beta^k$ is MAW of T, $\beta^{k'} \in \mathsf{D}_1$ with $k' \geq k$, since β^k must occurs in T. In addition, $\alpha\beta^{k-1} \in \mathcal{L}_\beta$ since $\alpha\beta^{k-1}$ occurs in T, and $\alpha\beta^k$ does not occurs in T. Therefore, $B \Rightarrow A$ is true. □

Using this proof, we can compute Type 5-1 MAW. Since each Type 5-1 MAW has a corresponding element of $\mathcal{L}_\beta$, for all elements of all D_3, we just verify whether the corresponding candidate is a MAW or not by D_1. Each verification costs $O(1)$ time, which leads to $O(m)$ time in total. The algorithm 5 show the procedure of enumerating all Type 5-1 MAW in this way.

Algorithm 4 Compute all $\mathsf{Exp}(\alpha u\beta)$ for D_4

```
Input: $\mathsf{tRLESA}, \mathsf{tRLELCP}$, initialized trie $SP$
Output: all $\mathsf{Exp}(\alpha u\beta)$ with $|\mathsf{Exp}(\alpha u\beta)| \geq 2$
 1: for each $i$ from $1$ to $m$ do
 2:     $x = \mathsf{tRLESA}[i]$
 3:     $i' = SP[a_{x-1}r_x]$
 4:     $x' = \mathsf{tRLESA}[i']$
 5:     for each $j$ with $\mathsf{LCP}(\mathsf{tRLE}(x), \mathsf{tRLE}(x')) \leq j \leq \mathsf{tRLELCP}[i]$ do
 6:         if $\mathsf{Exp}(\mathsf{Br}(x, j)) = \phi$ then
 7:             add $\mathsf{Br}(\mathsf{tRLESA}[i-1]-1, j+2)$ to $\mathsf{Exp}(\mathsf{Br}(x, j))$
 8:         end if
 9:         add $\mathsf{Br}(x-1, j+2)$ to $\mathsf{Exp}(\mathsf{Br}(x, j))$
10:     end for
11:     $SP[a_{x-1}r_x] = i$
12: end for
```

We can prove the case of Type 5-2 MAWs using a similar application.

We can represent Type 5-3 MAWs in the form of $u = \alpha\gamma^{k_1}u'\beta^{k_2}$ with $k_1 \geq 2, k_2 \geq 2$. u' might be ϵ. For some following proofs, we define two new array for each $\mathsf{Exp}(\gamma u'\beta)$. First, Let $H^{LR}(i)$ be the maximum number of E^{LR}_{max} such that $e_L \geq i$, and second, let $H^{RL}(i)$ be the maximum number of E^{RL}_{max} such that $e_R \geq i$. Then the following lemma holds.

Lemma 3.39. *Proposition $A_1 \wedge A_2$ equals to B.*

- *A_1: There are a node t that satisfies $E^{LR}_{max}(t) = k_2 - 1, c_L(t') = \alpha, e_L(t') = k_1$ in* $\mathsf{Exp}(\gamma u'\beta)$.
- *A_2: $H^{RL}(k_2) \geq k_1$ in* $\mathsf{Exp}(\gamma u'\beta)$.
- *B: $\alpha\gamma^{k_1}u'\beta^{k_2}$ is MAW of T.*

Proof. If there is a node t that satisfies $E^{LR}_{max}(t) = k_2 - 1, c_L(t') = \alpha, e_L(t') = k_1$ in $\mathsf{Exp}(\gamma u'\beta)$, $\alpha\gamma^{k_1}u'\beta^{k_2-1}$ occurs in T and $\alpha\gamma^{k_1}u'\beta^{k_2}$ does not occurs in T. If $H^{RL}(k_2) \geq k_1$, $\gamma^{k_1}u'\beta^{k_2}$ occurs in T. Therefore, $A_1 \wedge A_2 \Rightarrow B$ is true.

Algorithm 5 Compute all Type 5-1 MAWs

```
Input: $\mathsf{D}_1, \mathsf{D}_3$
Output: All Type 5-1 MAWs are added to $\mathsf{D}_5$
1: for each $c$ in $\Sigma$ occurs in $T$ do
2:     for each element $\alpha c^k$ with $\mathcal{L}_c$ do
3:         if $c^{k+1}$ occurs in $T$ then
4:             add $\alpha c^{k+1}$ to $\mathsf{D}_5$
5:         end if
6:     end for
7: end for
```

If $\alpha\gamma^{k_1}u'\beta^{k_2}$ is MAW of T, $\alpha\gamma^{k_1}u'\beta^{k_2-1}$ and $\gamma^{k_1}u'\beta^{k_2}$ occurs in T. It means that There are a node that satisfies $(E^{LR}_{max}, c_L, e_L) = (k_2-1, \alpha, k_1)$ in $\mathsf{Exp}(\gamma u'\beta)$. Furthermore, $H^{RL}(k_2) \geq k_1$ holds since $\gamma^{k_1}u'\beta^{k_2}$ occurs in T. Therefore, $B \Rightarrow A_1 \wedge A_2$ is true. $\square$

Using this proof, we can compute Type 5-3 MAW. Since each Type 5-3 MAW has a corresponding node in fixed D_4, for all elements of all D_4, we just verify whether the corresponding candidate is a MAW or not by H^{RL}. Each verification costs $O(1)$ time, which leads to $O(m)$ time in total. The algorithm 6 show the procedure of enumerating all Type 5-3 MAW in this way.

We can prove the case of Type 5-4 using a similar application.

We can represent Type 5-5 MAW in the form of $u = \alpha^{k_1}u'\beta^{k_2}$ with $k_1 \geq 2, k_2 \geq 2$. u' might be ϵ. Then the following lemma holds.

Lemma 3.40. *Proposition $A_1 \wedge A_2$ equals to B.*

- *A_1: $H^{LR}(k_1) = k_2 - 1$ in $\mathsf{Exp}(\alpha u'\beta)$*
- *A_2: $H^{RL}(k_2) = k_1 - 1$ in $\mathsf{Exp}(\alpha u'\beta)$*
- *B: $\alpha^{k_1}u'\beta^{k_2}$ is a MAW of T.*

Proof. If proposition A_1 is true, $\alpha^{k_1}u'\beta^{k_2}$ does not occur in T, and $\alpha^{k_1}u'\beta^{k_2-1}$ occurs in T. If proposition A_2 is true, $\alpha^{k_1}u'\beta^{k_2}$ does not occur in T, and $\alpha^{k_1-1}u'\beta^{k_2}$ occurs in T. Therefore, $A_1 \wedge A_2 \Rightarrow B$ is true.

Algorithm 6 Compute all Type 5-3 MAWs

Input: D_4, H^{LR}

Output: All Type 5-3 MAW are added to D_5

```
1:  for each $\gamma u'\beta$ with $|\mathsf{Exp}(\gamma u'\beta)| \geq 2$ do
2:      for each node $t \in V_L$ in $\mathsf{Exp}(\gamma u'\beta)$ do
3:          $\alpha = c_L(t)$
4:          $k_1 = e_L(t)$
5:          $k_2 = E^{LR}_{max}(t) + 1$
6:          if $H^{RL}(k_2) \geq k_1$ then
7:              add $\alpha\gamma^{k_1}u'\beta^{k_2}$ to $\mathsf{D}_5$
8:          end if
9:      end for
10: end for
```

If proposition B is true, $\alpha^{k_1}u'\beta^{k_2}$ does not occur in T, which leads to $H^{LR}(k_1) < k_2$, $H^{RL}(k_2) < k_1$ in $\mathsf{Exp}(\alpha u'\beta)$ However, $\alpha^{k_1}u'\beta^{k_2-1}$ occurs in T, which leads to $H^{LR}(k_1) \geq k_2 - 1$. Furthermore, $\alpha^{k_1-1}u'\beta^{k_2}$ occurs in T, which leads to $H^{RL}(k_2) \geq k_1 - 1$. Therefore, $B \Rightarrow A_1 \wedge A_2$ is true. $\square$

For this proof, we can compute Type 5-5 MAW. Since each Type 5-5 MAW has a corresponding H^{LR} in fixed D_4, for all H^{LR} of all D_4, we just verify whether the corresponding candidate is a MAW or not by H^{RL}. Each verification costs $O(1)$ time, which leads to $O(m)$ time in total.

To summarize, we have proven the following:

Lemma 3.41. *We can construct* D_5 *in* $O(m)$ *time.*

Algorithm 7 Compute all Type 5-5 MAWs

Input: $\mathsf{D}_4, H^{LR}, H^{RL}$

Output: All Type 5-5 MAWs are added to D_5

```
1: for each $\alpha u' \beta$ with $|\mathsf{Exp}(\alpha u' \beta)| \geq 2$ do
2:     for each $t$ such that $t = e_L(k)$ for some $k$ do
3:         $t' = H^{LR}(t) + 1$
4:         if $t = H^{RL}(t') + 1$ then
5:             add $\alpha^t u' \beta^{t'}$ to $\mathsf{D}_5$
6:         end if
7:     end for
8: end for
```

3.3 MAWs for sliding window on RLE strings

In this section we analyze a more advanced setting, which is a combination of the two models, the sliding window and Run-Length Encoding. In this section, We show some lower bounds of the amount of change that happens by adding one additional run to the end of the sliding window. Without loss of generality, we assume that we add run of `0` to the sliding window.

Table 3.1 shows a summary of the changes of all 8 types MAWs we mentioned in the previous section, when the current window T is extended by a new run. Note that we cannot grasp the fixed size of window by the total length that we defined in previous section. Without loss of generality, we assume that the new run is $\mathtt{0}^p$, and we assume that $m = R(T)$ is the sliding window size, in other words m is the number of runs in the window. For that, we define d to be the length of the string in this section, in other words $d = |T|$.

bounds	lower bound		upper bound	
added substrings	$\mathtt{0}$	$\mathtt{0}^p$	$\mathtt{0}$	$\mathtt{0}^p$
Type-A	1	1	1	1
Type-B	σ'	$p\sigma'$	σ'	$\Omega(\min(p\sigma', d))$
Type-C	$d-1$	$d-1$	$d-1$	$d-1$
Type-1	1	1	1	1
Type-2	$2\sigma'$	$2\sigma'$	$2\sigma'$	$2\sigma'$
Type-3	$\Omega(d)$	$\Omega(d)$	$O(d)$	$O(d)$
Type-4	$\Omega(m)$	$\Omega(m)$	$O(m^2)$	$O(m^2)$
Type-5	$\Omega(m)$	$\Omega(m)$	$O(m)$	$O(m)$

Table 3.1: The upper and lower bounds of the increase of MAWs in each type in the RLE string in the sliding window. Note that we do not count the deleted MAWs by a shift since it is at most one.

3.3.1 Bounds for the number of MAWs of Type-A, B and C

Lemma 3.42. $|\mathcal{M}_A(T\mathtt{0}^p) \setminus \mathcal{M}_A(T)| \leq 1$.

Proof. any MAW in $|\mathcal{M}_A(T\mathtt{0}^p) \setminus \mathcal{M}_A(T)|$ is the form of $\mathtt{0}^x$, and there are at most one value which fits x. We define that $\mathtt{0}^v$ is the longest suffix run of `0`. v might be 0. For any x with $x \leq p + v$, $\mathtt{0}^x$ is not a MAW since it occurs in $T\mathtt{0}^p$. For any x with $x \geq p + v + 2$, $\mathtt{0}^x$ is not a

MAW in $|\mathcal{M}_A(T\mathtt{0}^p) \setminus \mathcal{M}_A(T)|$ since $\mathtt{0}^{x-1}$ does not occur in $T\mathtt{0}^p$, or it is already in $|\mathcal{M}_A(T)|$. Thus $\mathtt{o}^{p+v+1}$ is the unique MAW in $|\mathcal{M}_A(T\mathtt{0}^p) \setminus \mathcal{M}_A(T)|$. □

Lemma 3.43. *There is a binary string T that satisfies $|\mathcal{M}_B(T\mathtt{0}^p) \setminus \mathcal{M}_B(T)| \in \Omega(p)$.*

Proof. Consider string $T = \mathtt{0}^p \cdot \mathtt{1}$. Then the set of Type-B MAWs for only $T\mathtt{0}^p$ is a superset of the set

$$\{\mathtt{1}\mathtt{0}^i\mathtt{1} \mid 1 \leq i \leq p\}.$$

Therefore, $|\mathcal{M}_B(T\mathtt{0}^p) \setminus \mathcal{M}_B(T)| \geq p \in \omega(m)$. □

Lemma 3.44. *There is a string T that satisfies $|\mathcal{M}_B(T\mathtt{0}^p) \setminus \mathcal{M}_B(T)| \in \Omega(\min(p\sigma', d))$, where σ' is the number of distinct characters occurring in T.*

Proof. Consider string $T = \mathtt{0}^p\mathtt{1} \cdot \mathtt{0}^p\mathtt{2} \cdots \mathtt{0}^p\sigma'$, Then the set of Type-B MAWs for only $T\mathtt{0}^p$ is a superset of the set

$$\{\sigma'\mathtt{0}^i c \mid 1 \leq i \leq p, c \in \{\mathtt{1}, \mathtt{2}...\sigma'\}\}.$$

Therefore, it means $|\mathcal{M}_B(T\mathtt{0}^p) \setminus \mathcal{M}_B(T)| \geq p\sigma \in \Omega(p\sigma')$. Since $p = \frac{n}{\sigma'}$, this string also satisfies $|\mathcal{M}_B(T\mathtt{0}^p) \setminus \mathcal{M}_B(T)| \geq p \cdot \sigma' \in \Omega(d)$. □

Note that we have a simple upper bound $|\mathcal{M}_B(T\mathtt{0}^p) \setminus \mathcal{M}_B(T)| \in O(p\sigma')$ by Lemma 3.3, which now turns out to be tight.

Lemma 3.45. $|\mathcal{M}_C(T\mathtt{0}^p) \setminus \mathcal{M}_C(T)| \leq |T|$.

Proof. Consider a pair of two distinct MAWs $a_1u_1\mathtt{0}^{x_1}$ and $a_2u_2\mathtt{0}^{x_2}$ in $\mathcal{M}_C(T\mathtt{0}^p) \setminus \mathcal{M}_C(T)$. Without loss of generality, let $x_2 \geq x_1 \geq 1$. By definition, $a_1u_1\mathtt{0}^{x_1-1}$ and $a_2u_2\mathtt{0}^{x_2-1}$ occur in T. We define $f(a_iu_i\mathtt{0}^{x_i-1}) = h_i$ with $i = \{1, 2\}$ such that $T[h_i - |a_iu_i|..h_i + x_i - 1] = a_iu_i\mathtt{0}^{x_i-1}$.

We prove that $h_1 \neq h_2$. If $h_1 = h_2$:

- If $|a_1u_1| < |a_2u_2|$, $u_2\mathtt{0}^{x_2}$ has $a_1u_1\mathtt{0}^{x_1}$ as its substring, which contradicts that $a_1u_1\mathtt{0}^{x_1}$ is a MAW of $T\mathtt{0}^p$. Figure 3.14 illustrates the contradiction briefly.
- If $|a_1u_1| > |a_2u_2|$, u_1 has a_2u_2 as its substring, which contradicts that $a_2u_2\mathtt{0}^{x_2}$ is a MAW since $a_2u_2\mathtt{0}^{x_2}$ occurs in $T\mathtt{0}^p$ as a suffix. Figure 3.14 illustrates the contradiction briefly.

Therefore, $h_1 \neq h_2$. It holds for any pairs of MAW in $|\mathcal{M}_C(T\mathtt{0}^p) \setminus \mathcal{M}_C(T)|$. It means that there is a mono-morphism from any position of T to all MAWs in $\mathcal{M}_C(T\mathtt{0}^p) \setminus \mathcal{M}_C(T)$. □

This lemma can be regarded as an upward compatibility to Lemma 3.4.

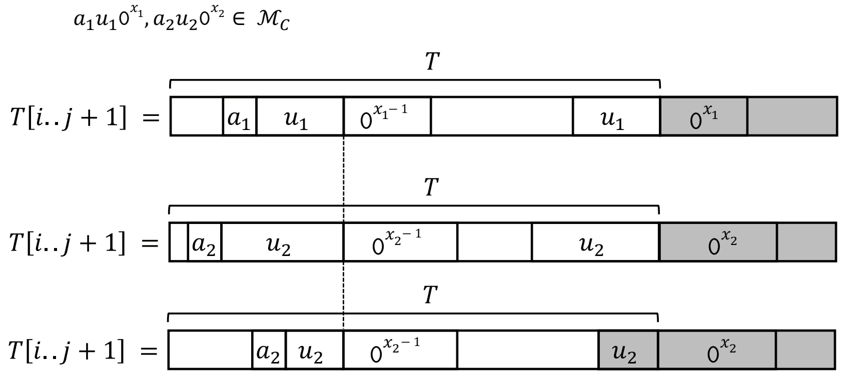


Figure 3.14: Contradiction of Type-C MAW. $a_1u_1\mathtt{0}^{x_2-1}$ occurs in T. If $|a_1u_1| > |a_2u_2|$, $a_1u_1\mathtt{0}^{x_2-1}$ has $a_2u_2\mathtt{0}^{x_2-1}$ as its substring. If $|a_1u_1| < |a_2u_2|$, $u_2\mathtt{0}^{x_2}$ has $a_1u_1\mathtt{0}^{x_2}$ as its substring.

3.3.2 Bounds for the number of MAWs of Type-1 to 5

Here, we show how the sliding window affects Type-1 to Type-5 MAW. To start, we explain trivial upper and lower bounds by a shift in each of Type-1 and Type-2.

- The upper and lower bounds on the amount of change for Type-1, are identically the same as for Type-A since both are unary strings and any other types (B, C, 2, 3, 4, 5) are not unary strings.

- The upper bound on the amount of increase for Type-2 is at most $2\sigma'$. Any Type-2 MAW is a bigram, and any Type-2 MAW that is generated by a shift, must include the added character 0 in its substring. The number of such bigrams is at most $2\sigma'$, where σ' is the number of characters occurring in current window. Moreover, we can easily see that it is also a tight bound by Lemma 3.16, which is the case that the amount of increase for Type-2 is just $2\sigma'$.

Therefore, from now on we focus on the the amount of change for Type-3 to 5.

Lemma 3.46. $|\mathcal{M}_3(T\mathtt{0}^p) \setminus \mathcal{M}_3(T)| \in O(d)$.

Proof. Let t be the rightmost character which is not 0. Any MAW of Type-3 is a form of ab^ic with $a, b, c \in \Sigma, a \neq b, b \neq c, i \geq 1$. Then, for any MAW $ab^ic \in \mathcal{M}_3(T\mathtt{0}^p) \setminus \mathcal{M}_3(T)$, the following two rules hold.

- If ab^i occurs in T, b^ic does not occurs in T. Then, such MAW is a form of $ab^i\mathtt{0}$ since b^ic overlaps between T and $\mathtt{0}^p$. therefore, such MAW belongs to $\mathcal{M}_3(T\mathtt{0}) \setminus \mathcal{M}_3(T)$. Since

$|\mathcal{M}_3(T\mathtt{0}) \setminus \mathcal{M}_3(T)| \in O(d)$ holds by lemma 3.5, the number of such MAW is at most $O(d)$.

- If $b^i c$ occurs in T, ab^i does not occurs in T. Then, such MAW is a form of $t\mathtt{0}^i c$ since ab^i overlaps between T and $\mathtt{0}^p$. For each element $\mathtt{0}^{p_2}c'$ of $\mathcal{R}_\mathtt{0}$, The number of candidate of such MAW is at most p_2 (a form of $t\mathtt{0}^{p_3}c'$ with $p_3 \leq p_2$). Since any two distinct element of $\mathcal{R}_\mathtt{0}$ never overlap, $\|\mathcal{R}_\mathtt{0}\| \in O(d)$. Therefore, the number of such MAW is at most $O(d)$.

Finally, $|\mathcal{M}_3(T\mathtt{0}^p) \setminus \mathcal{M}_3(T)| \in O(d)$ holds. □

Lemma 3.47. *There is a binary string T that satisfies $|\mathcal{M}_3(T\mathtt{0}) \setminus \mathcal{M}_3(T)| \in \Omega(d)$.*

Proof. Consider string $T = \mathtt{01}^i$, then the set of Type-3 MAWs for only $T\mathtt{0}$ is a superset of the set

$$\{\mathtt{01}^i\mathtt{0} \mid 1 \leq i < d-2\}.$$

Therefore, $|\mathcal{M}_3(T\mathtt{0}) \setminus \mathcal{M}_3(T)| \geq d-2 \in \Omega(d)$. □

Lemma 3.48. *There is a binary string T that satisfies $|\mathcal{M}_4(T\mathtt{0}) \setminus \mathcal{M}_4(T)| \in \Omega(m)$, where $m = R(T)$.*

Proof. Consider string $T = \mathtt{10} \cdot (\mathtt{1}^2\mathtt{0}^2)^p \cdot \mathtt{1}$, Then the set of Type-4 MAWs for only $T\mathtt{0}$ is a superset of the set

$$\{\mathtt{10}(\mathtt{1100})^i\mathtt{10} \mid 0 \leq i < p\}.$$

Since $m = 2p+3$, $|\mathcal{M}_4(T\mathtt{0}) \setminus \mathcal{M}_4(T)| \geq p-1 \in \Omega(p) = \Omega(m)$. □

Lemma 3.49. *There is a binary string T that satisfies $|\mathcal{M}_5(T\mathtt{0}) \setminus \mathcal{M}_5(T)| \in \Omega(m)$, where $m = R(T)$.*

Proof. Consider string $T = \mathtt{11} \cdot (\mathtt{01})^p \cdot \mathtt{0}$, Then the set of Type-5 MAWs for only $T\mathtt{0}$ is a superset of the set

$$\{\mathtt{1}^2(\mathtt{10})^i\mathtt{0}^2 \mid 0 \leq i < p\}.$$

Since $m = 2p+2$, $|\mathcal{M}_5(T\mathtt{0}) \setminus \mathcal{M}_5(T)| \geq p-1 \in \Omega(p) = \Omega(m)$. □

Note that we have a simple upper bound $|\mathcal{M}_5(T\mathtt{0}) \setminus \mathcal{M}_5(T)| \in O(m)$ by lemma 3.22. Therefore, lemma 3.49, 3.22 are both tight bounds.

Chapter 4

Grammar Index By Induced Suffix Sorting

GCIS[94, 93] is based on the idea from the famous SAIS algorithm [92] that builds the suffix array of an input string in linear time. Recently, it is shown that GCIS has a locally consistent parsing property similar to the ESP-index [77], the SE-index [89], and grammar-based indexing structures based on GCIS have been proposed [2, 35].

4.1 Constructing the GCIS grammar

Let T be the string of length n over an integer alphabet $\Sigma = \{\mathtt{1}, \dots, \sigma\}$. Let $\Gamma = \{\sigma+1, \dots, \sigma+|\Gamma|\}$ be the set of non-terminal symbols. For strings x, y over Σ or Π, we write $x \prec y$ iff x is lexicographically smaller than y.

First we explain how the GCIS algorithm constructs its grammar from the input string. For any text position $1 \leq i \leq |T|$, position i is of type L if $T[i..|T|]$ is lexicographically larger than $T[i+1...|T|]$, and it is of type S otherwise. For any $2 < i < |T|$, we call position i an *LMS* (*LeftMost S*) position if i is of type S and $i-1$ is of type L. For convenience, we append a special character $\$$ to T which does not occur elsewhere in T, and assume that positions 1 and $|T\$|$ are LMS positions.

Let $i_1, \dots, i_{z+1}$ be the sequence of the LMS positions in T sorted in increasing order. Let $D_j = T[i_j..i_{j+1} - 1]$ for any $1 \leq j \leq z$. D_j is the j-th LMS-substrings. When $z \geq 2$, then $T = D_1, \dots, D_z$ is called the GCIS-parsing of T.

We assign each computed factor $F_j^{(h)}$ a non-terminal $X_j^{(h)}$ such that $X_j^{(h)} \rightarrow F_j^{(h)}$, but omit the delimiter $\mathtt{\$}$. The order of the non-terminals $X_j^{(h)}$ is induced by the lexicographic order of their respective LMS-substrings. We now use the non-terminals instead of the lexicographic ranks in the recursive steps. If we set $X^{(\tau_T)} \rightarrow T^{(\tau_T)}$ as the start symbol, we obtain a context-free grammar $\mathcal{G}_T := (\Sigma, \Gamma, \pi, X^{(\tau_T)})$, where Γ is the set of non-terminals and a function $\pi : \Gamma \to (\Sigma \cup \Gamma)^+$ that applies production rules. For simplicity, we say that $\pi(c) = c$ for $c \in \Sigma$. Let g denote the sum of the right hand sides of all grammar rules. We say that a non-terminal ($\in \Gamma$) or a character ($\in \Sigma$) is a *symbol*, and denote the set of characters and non-terminals with $\mathcal{S} := \Sigma \cup \Gamma$. We understand π also as a string morphism $\pi : \mathcal{S}^* \to \mathcal{S}^*$ by applying π on each symbol of the input string. This allows us to define the *expansion* $\pi^*(X)$ of a symbol X, which is the iterative application of π until obtaining a string of characters, i.e., $\pi^*(X) \subset \Sigma^*$. Since $\pi(X)$ is deterministically defined, we use to say *the right hand side of* X for $\pi(X)$.

Lemma 4.1 ([94]). *The GCIS grammar $\mathcal{G}_T$ can be constructed in $O(n)$ time. $\mathcal{G}_T$ is* reduced, *meaning that we can reach all non-terminals of* Γ *from* $X^{(\tau_T)}$.

Although [94] did not state their model of computation, it is easy to verify that 4.1 holds in the pointer machine model.

$\mathcal{G}_T$ can be visualized by its derivation tree, which has $X^{(\tau_T)}$ as its root, and each production $X_k^{(h)} \to X_i^{(h-1)} \cdots X_j^{(h-1)}$ defines a node $X_k^{(h)}$ having $X_i^{(h-1)}, \ldots, X_j^{(h-1)}$ as its children. The height of the derivation tree is $\tau_T = O(\lg n)$ because the number of LMS substrings of $T^{(h)}$ is at most half of the length of $T^{(h)}$ for each recursion level h. The leaves of the derivation tree are the terminals at height 0 that constitute the characters of the text T. Reading the nodes on height $h \in [0..\tau_T - 1]$ from left to right gives $T^{(h)}$ with $T^{(0)} = T$. Note that we use the derivation tree only as a conceptional construct since it would take $O(n)$ words of space. Instead, we merge (identical) subtrees of the same non-terminal together to form a directed acyclic graph (DAG), which we call DAG. We let each non-terminal store a pointer to its parent such that we can climb up from every non-terminal to the DAG root $T^{(\tau_T)}$.

By construction, each non-terminal appears exactly in one height of DAG. More precisely, π maps a non-terminal on height $h > 1$ to a string of symbols on height $h - 1$. Hence, the grammar is acyclic.

Unfortunately, there are strings with $g = \Omega(n)$, so for a particular input, the grammar does not compress at all: Let $T = \prod_{i=0}^{m} \mathtt{a}^i\mathtt{b} = \mathtt{b} \cdot \mathtt{ab} \cdot \mathtt{aab} \cdot \mathtt{aaab} \cdots$ with $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\mathtt{a} < \mathtt{b}$. Then we have the rules $X_i^{(1)} \to \mathtt{a}^i\mathtt{b}$ for $i \in [0..m]$, with $X^{(2)} \to \prod_{i=0}^{m} X_i^{(1)}$ being the production of the start symbol $X^{(2)}$. Hence, $\tau_T = 2$ and $g = |T| + |\pi(X^{(2)})| = |T| + m + 1$. Nevertheless, the experiments in 4.5 show that the grammar is suitable for most highly-repetitive text collections.

4.2 GCIS Index

In what follows, we want to show that we can augment $\mathcal{G}_T$ with auxiliary data structures for answering locate. Our idea stems from the classic pattern matching algorithm with the suffix tree [55, APL1]. The key difference is that we search the core of a pattern in the right hand sides of the production rules.

For that, we make use of the generalized suffix tree GST built upon $R := \prod_{X \in \Gamma} \pi(X)\mathtt{\$}$, i.e., the right hand sides of all production rules, for a special delimiter symbol $\mathtt{\$}$ being smaller than all symbols. Since we have a budget of $O(g)$ words, we can afford to use a plain pointer-based tree topology. Each leaf λ stores a pointer to the non-terminal $X^{(h)}$ and an offset ℓ such that the path from the root to λ has $\pi(X^{(h)})[\ell..]$ as its prefix. We need the following operation on GST:

- $lca(u, v)$: gives the lowest common ancestor of two nodes u and v. We can augment GST with the data structure of [16] in linear time and space in the number of nodes of GST. This data structure answers lca in constant time.

- $child(u, c)$: gives the child of the GST node u connected to u with an edge having a label starting with $c \in \Gamma$. Our GST implementation answers child in $O(\lg |\mathcal{S}|)$ time. For that, each node stores the pointers to its children in a binary search tree with the first symbol of each connecting edge as key.

- $string\_depth(u)$: returns the string depth of a node v, i.e., the length of its string label, which is the string read from the edge labels on the path from the root to v. We can compute and store the string depth of each node during its construction.

The operation child allows us to compute the **locus** of a string S, i.e., the highest GST node u whose string label has S as a prefix, in $O(|S| \lg |\mathcal{S}|)$ time. For each $\pi(X)$, we augment the locus u of $\pi(X)\mathtt{\$}$ with a pointer to X such that we can perform

- $lookup(S)$: returns the non-terminal X with $\pi(X) = S$ or an invalid symbol $\perp$ if such an X does not exist. The time is dominated by the time for computing the locus of S.

The leaves of GST are connected with a linked list. Each internal node v stores a pointer to the leftmost leaf in the subtree rooted at v. With that we can use the function

- $lce(X, Y, i, j)$: returns the longest common prefix of $\pi(X)[i..]$ and $\pi(Y)[j..]$ for $X, Y \in \Gamma$ and $i \in [1..|\pi(X)|], j \in [1..|\pi(Y)|]$. We can answer $lce(X, Y, i, j)$ be selecting the leaves $X.P[i]$ and $Y.P[j]$, retrieve the LCA $lca(X.P[i], Y.P[j])$ of both leaves, and take its string depth, all in constant time. More strictly speaking, we return $\min(|\pi(X)[i..]|, |\pi(Y)[j..]|, \mathit{string\_depth}(\mathit{lca}(X.P[i], Y.P[j])))$, since the delimiter `$` is not a unique character, but appears at each end of $\pi(\cdot)$ in the underlying string R of GST.

Each rule $X \in \Gamma$ stores an array $X.P$ of $\pi(X)$ pointers to the leaves in GST such that the $X.P[i]$ points to the leaf that points back to X and has offset i (its string label has $\pi(X)[i..]$ as a prefix). Additionally, each rule X stores the length of $\pi(X)$, an array $X.L$ of all expansion lengths of all its prefixes, i.e., $X.L[i] := \sum_{j=1}^{i} |\pi^*(\pi(X)[j])|$, and an array $X.R$ of the lengths of the right hand sides of all its prefixes, i.e., $X.R[i] := \sum_{j=1}^{i} |\pi(\pi(X)[j])|$.

Complexity Bounds GST can be computed in $O(g \lg |\mathcal{S}|)$ time with Weiner's algorithm [107]. The grammar index consists of the GCIS grammar, GST built upon $g + |\Gamma|$ symbols, and augmented with an lca-data structure [16]. This all takes $O(g)$ space. Each non-terminal is augmented with an array $X.P$ of pointers to leaves and $X.L$ storing the expansion lengths of all prefixes of $\pi(X)$, which take again $O(g)$ space when summing over all non-terminals.

4.3 Pattern matching algorithm

Like [99, Sect. 2], our idea is to first find the occurrences of C in the text, and then try to extend all these occurrences to occurrences of P. However, we do not try to find the occurrences of P directly, but rather the non-terminals lowest in DAG containing a core C in their expansions that are large enough to contain the whole pattern P.

4.3.1 Cores

A core is a string of symbols C of the GCIS grammar $\mathcal{G}_P$ built on the pattern P, which has the property that all occurrences of C whose expansion can be extended to an occurrence of P are (not necessarily proper) substrings of the right hand sides of the production rules of $\mathcal{G}_T$. Put differently, given C consists of consecutive nodes on height $h \geq 0$ in the grammar tree, if there is an occurrence of C in the grammar tree that has at least two parent nodes on height $h + 1$, then the expansion of this occurrence of C does not lead to an occurrence of P. We qualify a core by the difference in the number of occurrences of P and C in the grammar tree. On the one hand, although a character $P[i]$ always qualifies as a core, the appearance of $P[i]$ in T is unlikely to be an evidence of an occurrence of P. (additionally, our algorithm would perform worse than a standard pattern matching with the suffix tree when resorting to single characters as cores). On the other hand, the non-terminal covering most of the characters of P might not be a core. Hence, we aim for the highest possible non-terminal, for which we are sure that it exhibits the core property.

Finding a Core We determine a core C of P during the computation of the GCIS grammar $\mathcal{G}_P$ of P. During this computation, we want to assure that we only create a new non-terminal for a factor F whenever $\mathit{lookup}(F) = \bot$; if $\mathit{lookup}(F) = X$, we borrow the non-terminal X from $\mathcal{G}_T$. By doing so, we ensure that non-terminals of $\mathcal{G}_P$ and $\mathcal{G}_T$ are identical whenever their right hand sides of their productions are equal. In detail, if we create the factors $P^{(h)} = F_1^{(1)} \cdots F_{z_h}^{(h)}$, we first retrieve $Y_i^{(h)} := \mathit{lookup}(P^{(h)})$ for each $i \in [2..z_h - 1]$. If one of the lookup-queries returns $\bot$, we abort since we can be sure that the pattern does not occur in T. That is because all non-terminals $Y_2^{(h)}, \ldots Y_{z_h-2}^{(h)}$ classify as cores. To see this, we observe that prepending or appending symbols to $P^{(h)}$ does not change the factors $F_2^{(h)}, \ldots, F_{z-1}^{(h)} =: C^{(h)}$.

Correctness We show that when prepending or appending characters to $F_1^{(h)} C^{(h)} F_{z_h}^{(h)}$, then the LMS factorization of the resulting string still keeps the symbols of $C^{(h)}$ within a single factor. What we show is that (a) we cannot change the type of any position $C^{(h)}[i]$ in $C^{(h)}$ to $\mathtt{S}^*$ on extending $C^{(h)}$ and (b) after appending a character, the last position of $C^{(h)}$ cannot be $\mathtt{S}^*$. Showing both claims makes it impossible to let a new factor start at $C^{(h)}[i]$.

Prepending The type of a position ($\mathtt{S}$ or $\mathtt{L}$) depends only on its succeeding position, and hence prepending cannot change the type of a position in $C^{(h)}$.

Appending If $F_{z_h}^{(h)}$ is non-empty, then appending characters can either prolong $F_{z_h}^{(h)}$ or create a new factor since $F_{z_h}^{(h)}$ starts with $\mathtt{S}^*$ and therefore appending cannot change $C^{(h)}$. If $F_{z_h}^{(h)}$ is empty, then $C^{(h)}$ ends with P, and as a border case, the last position of $C^{(h)}$ is $\mathtt{S}^*$. In that case, when appending a symbol smaller than $P[m]$ to $F_1^{(h)} C^{(h)}$ changes the type of the last position of $C^{(h)}$ to $\mathtt{L}$. If we append a symbol larger than $P[m]$, then the last position of $C^{(h)}$ becomes $\mathtt{S}$, but does not become $\mathtt{S}^*$ since $P^{(\tau_P)}[j_k] > P^{(\tau_P)}[j_{k-1}]$ due to construction (otherwise $F_{z_h}^{(h)}$ would not be empty).

The construction of $\mathcal{G}_P$ iterates the LMS factorization until we are left with a string of symbols $P^{(\tau_P)}$ whose factorization consists of at most two factors. In that case, we split $P^{(\tau_P)}$ into three factors $C_p C C_s$ with C_p and C_s possible empty, and defined as follows:

- If the LMS factorization consists of two non-empty factors $F_1 \cdot F_2$, then C_p is F_1.

- Given $P^{(\tau_P)} = (P^{(\tau_P)}[j_1])^{c_1} \cdots (P^{(\tau_P)}[j_k])^{c_k}$ is the run-length-encoded representation of P with $1 = j_1 < \ldots < j_k = |P^{(\tau_P)}|$, $c_{j_i} \geq 1$ for $i \in [1..k]$, and $P[j_i] \neq P[j_{i+1}]$ for $i \in [1..k-1]$, we set $C_s \leftarrow (P^{(\tau_P)}[j_k])^{c_k}$ if $P^{(\tau_P)}[j_k] < P^{(\tau_P)}[j_{k-1}]$.
- In the other cases, C_p and/or C_s are empty.

In total, there are symbols $P^{(1)}, \ldots, P^{(\tau_P-1)}$ and $S^{(1)}, \ldots, S^{(\tau_P-1)}$ such that $P = \pi(F_1^{(1)} \cdots F_1^{(\tau_P-2)} P^{(1)} \cdots P^{(\tau_P-2)} C S^{(\tau_P-2)} \cdots S^{(1)} F_{z_{\tau_P-2}}^{(\tau_P-2)} \cdots F_{z_1}^{(1)})$, and $P^{(h)}$, $S^{(h)}$ are symbols at height $h \in [0..\tau_P - 2]$ that are cores of P, while $F_1^{(h)}$ and $F_{z_h}^{(h)}$ are single factors containing symbols of height $h-1$.

4.3.2 Pattern matching with Generalized Suffix Tree (GST)

Having C, we now switch to GST and use it to find all non-terminals of $\mathcal{G}_T$ whose right hand side contains C. Let $\mathrm{occ}_C \in O(g)$ denote the number of occurrences of C in the right hand sides of all production rules of $\mathcal{G}_T$. Our first task is to find, for each occurrence O of C in DAG, O's lowest ancestor whose expansion is large enough to not only contain O but also P by extending O to its left and right side. For that, we first spot the non-terminals containing C, and then climb up DAG if necessary: We first compute the locus v of C in $O(|C| \lg |\mathcal{S}|)$ time via child. Subsequently, we take the pointer to the leftmost leaf in the subtree rooted at v, and then process all leaves in this subtree by using the linked list of leaves. For each such leaf λ, we compute a path in form of a list λ_L from the non-terminal containing C on its right hand side up to an ancestor of it that has an expansion large enough to cover P if we would expand the contained occurrence of C to P. We do so as follows: Each of these leaf stores a pointer to a non-terminal X and a starting position i such that we know that $\pi^*(X)[i..]$ starts with $\pi^*(C)$. By knowing the expansion lengths $X.L[|\pi(X)|]$, $X.L[i-1]$, and $|\pi^*(C)|$, we can judge whether the expansion of X has enough characters to be able to extend its occurrence of C to P. If it has enough characters, we put (X, i) onto λ_L such that we know that $\pi^*(X)[X.L[i-1]+1..]$ has C as a prefix. If X does not have enough characters, we exchange C with X and recurse on finding a non-terminal with a larger expansion. By doing so, we visit $O(\lg m)$ non-terminals per occurrence of C in the right hand sides of $\mathcal{G}_T$ since the GCIS grammar of the pattern has a depth of $O(\lg m)$. Given that occ_C denotes this number of occurrences of C in DAG, we perform all operations in $O(\mathrm{occ}_C \lg m \lg |\mathcal{S}|)$ time, since for each recursion step, we query child.

The previous step computes a path $(Y^{(h)}, \ldots, Y^{(\tau_P)})$ in DAG for each occurrence of C and an offset $o^{(\tau_P)}$ such that $Y^{(\tau_P)}[o^{(\tau_P)}] = C$.

By construction, these paths cover all occurrences of C in DAG. Note that we process the DAG node $Y^{(\tau_P)}$ (but for different offsets $o^{(\tau_P)}$) as many times as C occurs in its right hand side). In what follow, we try expand the occurrence of C captured by $Y^{(\tau_P)}$ and $o^{(\tau_P)}$ to an occurrence of P. Naively, we would walk down from $Y^{(\tau_P)}[o^{(\tau_P)}]$ to the character level and extend the substring $\pi^*(C)$ in both directions by character-wise comparison with P. However, this would take $O(\mathrm{occ}_C m \lg m)$ time since each non-terminal $Y^{(\tau_P)}$ is of height $O(\lg m)$, and the number of occurrences of C in DAG is occ_C. Our claim is that we can perform the computation in $O(m + \mathrm{occ}_C \lg m)$ time with the aid of lce and an amortization argument.

For that, we use the cores of P with $P = \pi(F_1^{(1)} \cdots F_1^{(\tau_P-2)} P^{(1)} \cdots P^{(\tau_P-2)} C S^{(\tau_P-2)} \cdots S^{(1)} F_{z_{\tau_P-2}}^{(\tau_P-2)} \cdots F_{z_1}^{(1)})$ according to 4.3.1, which allows us to use LCE queries in the sense that we can try to extend an occurrence of C with an already extended occurrence (that maybe does not match P completely). For the explanation, we only focus on extending all occurrences of C to the right to CC_s (the left side side is done symmetrically). We maintain an array D of length $\lg m$ storing triplets $(X^{(h)}, o_h, \ell_h)$ for each height $h \in [1..\lg m]$ such that the non-terminal $X^{(h)}$ with the respective offset o_h has the currently longest extension of length ℓ_h of its occurrence of C with CC_s at height h (more precisely, we mean $\pi^{(h-\tau_P)}(CC_s)$). By maintaining D, we can first query lce with the specific non-terminal in D, and then resort to plain symbol comparison. Given that we matched ℓ symbols, we descend to the child where the mismatch happens and recurse until reaching the character level of DAG. This works since by the core property the mismatch of a child means that there is a mismatch in the expansion of this child. Since a plain symbol comparison with matching symbols lets us exchange the currently used non-terminal in D with a longer one, we can bound (a) the total number of naive symbol matches to $O(m)$ and (b) the total number of naive symbol mismatches and LCE queries to $O(\mathrm{occ}_C \lg m)$.

In technical terms it works as follows: We start with D storing only invalid entries $\perp$ to indicate that there is no candidate on any height for LCE queries.

Now given a path $(Y^{(h)}, \ldots, Y^{(\tau_P)})$ in DAG for each occurrence of C and an offset $o^{(\tau_P)}$ such that $Y^{(\tau_P)}[o^{(\tau_P)}] = C$. Let $\ell \leftarrow 0$ be the number characters we matched $\pi^*(Y^{(\tau_P)})[o + \pi^*(C)..]$ with C_s so far, which is zero in the beginning. We scan the entries of D from $j = \tau_P$ to $j = 1$ while checking whether $D[h] = \perp$: Suppose that $D[j] = (\tilde{Y}^{(j)}, \tilde{o}^{(j)}, \tilde{\ell}^{(j)})$ is not empty, we compute $o^{(j-1)} := lce(\tilde{Y}^{(j)}, \tilde{o}^{(j)}, Y^{(j)}, o^{(j)})$, set $Y^{(j-1)} := Y^{(j)}[o^{(j-1)}]$ and $\ell = \ell + \tilde{Y}^{(o^{(j-1)})} - \tilde{Y}^{(j)}[\tilde{o}^{(j)}]$. If $D[j] = \perp$, we set $o^{(j-1)} = 0$ and $Y^{(j-1)} := Y^{(j)}[0]$. On reaching $j = 1$, we compare the right hand sides of $Y^{(1)}$ and its right siblings with C_s. On reaching a mismatch or finding C_s, we update the entries of D while climbing up path, and store the node $Y^{(h)}$ with the starting position of the respective occurrence of P relative to its grammar subtree in a list W. The traversal, the update of D, and LCE queries take $O(\lg m)$ time per occurrence of C, while the total number of naive symbol matches accumulate to $O(m)$ time.

Finding the Starting Positions It is left to compute the starting position in T of each occurrence captured by an element in W. We can do this similarly to computing the pre-order ranks in a tree: For each pair (X, ℓ), climb up DAG from X to the root while accumulating the expansion lengths of all left siblings of the nodes we visit (we can make use of $X.L$ for that). If this accumulated length is s, then $\ell + s$ is the starting position of the occurrence captured by (X, ℓ). However, this approach would cost $O(\lg n)$ time per element of W. Here, we use the amortization argument of [25, Sect. 5.2], which works if we augment, in a pre-computation step, each non-terminal X in Γ with (a) a pointer to its lowest ancestor Y_X that has X at least twice in its subtree, and (b) the lengths of the expansions of the left siblings of the child of Y_X being a parent of X or X itself. By doing so, when taking a pointer of a non-terminal X to its ancestor Y_X, we know that X has another occurrence in DAG (and thus there is another occurrence of P). Therefore, we can charge the cost of climbing up the tree with the amount of occurrences occ of the pattern.

Total time To sum up, we spent $O(m \lg |\mathcal{S}|)$ time for finding C, $O(\mathrm{occ}_C \lg m \lg |\mathcal{S}|))$ time for computing the non-terminals covering C, $O(m + \mathrm{occ}_C \lg m)$ time for reducing these non-terminals to W, and $O(\mathrm{occ})$ time for retrieving the starting positions of the occurrences of P in T from W. To be within our $O(g)$ space bounds, we can process each occurrence of C in DAG individually, and keep only D globally stored during the whole process. The total additional space is therefore $O(\lg m)$ for maintaining D and a path for each occurrence of C. So we finally obtain the claim of 1.1.

4.4 Implementation

The implementation deviates from theory with respect to the rather large hidden constant factor in the $O(g)$ words of space. We drop GST, and represent DAG with multiple arrays. For that, we first enumerate the non-terminals as follows:

The height and the lexicographic order induce a natural order on the non-terminals in Γ, which are ranked by first their height and secondly by the lexicographic order of their right hand sides, such that we can represent $\Gamma = \{X_1, \ldots, X_{|\Gamma|}\}$. By stipulating that all characters are lexicographically smaller than all non-terminals, we obtain the property is that $\pi(X_i) \prec \pi(X_{i+1})$ for all $i \in [1..|\Gamma| - 1]$. In the following, we first present a plain representation of DAG, then give our modified locate algorithm, and subsequently present a compressed version of DAG using universal coding.

The plain variant, which we call GCIS-nep, represents each symbols with a 32-bit integer. We use $R = \prod_{i=1}^{|\Gamma|} \pi(X_i)$ again, but omit the delimiters $\$$ separating the right hand sides. To find the right hand side of a non-terminal X_i, we create an array of positions $Q[1..|\Gamma|]$ such that $Q[i]$ points to the starting position of X_i's right hand side in R. Finally, we create an array $L[1..|\Gamma|]$ storing the lengths of the expansion $|\pi^*(X_i)|$ for each non-terminal X_i in $L[i]$. Due to the stipulated order of the symbols, the strings $R[Q[i]..Q[i+1]-1]$ are sorted in ascending order. Hence, we can evaluate $lookup(S)$ for a string S in $O(|S| \lg |\mathcal{S}|)$ time by a binary search.

Locate Our implementation follows theory, but deviates after computing the core C. We first compute $\mathcal{G}_P$, for which we use a binary search on the lexicographically sorted right hand side rules represented by R and Q to query with lookup for the existence of a non-terminal. Hence, we can determine the core C in the same time bound $O(\lg |\mathcal{S}|)$ as before. To find all non-terminals whose right hand sides contain C, we can use again a binary search if C_p is not empty: in that case we know that all occurrences of πC are prefixes in the right hand sides. Otherwise, we have to linearly scan the right hand sides of all non-terminals at level τ_P, which we can do cache-friendly since the right-hands of R are sorted by the height of their respective non-terminals. This takes $O(g)$ time. Finally, for extending the found occurrences of a core C to an occurrence of P, we follow the naive approach to descend DAG to the character level and compare the expansion with P character-wise, which results in $O(\mathrm{occ}_C |P| \lg m)$ time. The total costs are $O(g + |P|(\mathrm{occ}_C \lg m + \lg \mathcal{S}))$ time.

GCIS-uni To save space, we can leverage universal code to compress the right hand sides of the productions. First, we observe that the first symbols $F := \pi(X_1)[1], \ldots, \pi(X_{|\Gamma|})[1]$ form an ascending sequence, which we encode in Elias–Fano [37]. The rest of each right hand side $\pi(X_i)[2..]$ is represented in delta-coding, i.e., $\Delta[i][k] := \pi(X_i)[k] - \pi(X_i)[k-1]$ for $k \in [2..|\pi(X_i)|]$, and stored in Elias-γ code [38]. Figure 4.1 show how these coding works. Finally, like in the first variant, we store the expansion lengths of all non-terminals in L. Here, we separate L in a first part using one byte per entry, then two bytes per entry, and finally four bytes per entry. This is done by representing L by three arrays, and continue with filling the next array whenever we process a value that can no longer be stored in the current array.

Since Elias–Fano code supports constant time random access and Elias-γ supports constant time linear access, we can decode a non-terminal X_i by accessing $F[i]$ and then sequentially decode $\Delta[i]$. Hence, we can simulate GCIS-nep with this compressed version without sacrificing the theoretical bounds. We call the resulting index GCIS-uni.

4.5 Experiments

In the following we present an evaluation of our C++ implementation and different self-indexes for comparison, which are the FM-index [40][1], the ESP-Index [103] [2], and the r-index [47] [3]. All code has been compiled with `gcc-10.2.0` in the highest optimization mode `-O3`, except the code for LZ-index [4], which is not conform with newer compilers — the authors provided a docker script to create a virtual environment with the compiler `gcc-4.8.4`. We ran all our experiments on a Mac Pro Server with an Intel Xeon CPU X5670 clocked at 2.93GHz running Arch Linux.

Table 4.1 shows the summary of index size. GCIS-uni occupies less space than other indexes in terms of index size. For dataset DNA and COMMONCRAWL, FM-index produced a more compact index size than our GCIS-uni. On the other hand, FM-index was unable to compress the artificial data efficiently.

Tables 4.5 and 4.6 summarize construction time and peak memory during indexing. In terms of peak memory, both GCIS-nep and GCIS-uni outperform ESP-index. However, both GCIS-nep and GCIS-uni are slower than ESP-index in terms of the speed of indexing.

Table 4.1: Sizes of the used data sets and the indexes stored on disk. Sizes are in megabytes [MB].

input text	input size	GCIS-nep	GCIS-uni	ESP-index	FM-index	r-index
ENGLISH.001.2	104.857	14.784	7.489	10.464	46.981	14.389
DNA	403.927	527.553	327.852	297.001	216.153	2123.817
COMMONCRAWL	221.180	220.119	138.856	156.006	122.575	454.124
TM29	268.435	0.002	0.001	0.002	69.347	0.009
FIB41	267.914	0.001	0.001	0.001	71.305	0.007
RS.13	216.747	0.002	0.001	0.002	57.653	0.009
KERNEL	257.961	21.298	10.469	12.545	125.087	28.947
INFLUENZA	154.808	23.373	13.871	15.729	53.066	28.775
WORLD LEADERS	46.968	5.415	2.573	3.611	21.097	5.627
EINSTEIN.DE.TXT	92.758	1.139	0.428	0.697	40.291	1.1458

We can observe in Figures 4.2 and 4.3 that our indexes answer $locate(P)$ fast when P is sufficiently long or has many occurrences occ in T. In particular for ENGLISH.001.2, Our implementations are faster than FM-index and r-index when the pattern length reaches 10000 characters and more. At this time, the pattern grammar reached a height τ_P of almost six, which

[1] `https://github.com/mpetri/FM-Index`

[2] `https://github.com/tkbtkysms/esp-index-I`

[3] `https://github.com/nicolaprezza/r-index`

[4] `https://github.com/migumar2/uiHRDC`

is the height τ_T. The algorithm can extend an occurrence of a core to a pattern occurrence by checking only 80 - 100 characters.

However, when the pattern surpasses 5000 characters, the computation of $\mathcal{G}_P$ becomes the time bottleneck. With that respect, the ESP-index shares the same characteristic. encoding make slow down the location time by about 2 to 10 times approximately. Let us have a look at the dataset FIB41, which is linearly recurrent [36], a property from which we can derive the fact that a pattern that occurs at least once in T has actually a huge number of occurrences in T. There are almost 3,000,000 occurrence of patterns with a length of 100. Here, we observe that our indexes are faster than ESP-index.

ESP-index needs more time for locate than GCIS because GCIS can form a core that covers a higher percentage of the pattern than the core selected by ESP.

FM-index, and ESP-index with $|P| = 10$ take 100 seconds or more on average – we omitted them in the graph to keep the visualization clear.

Next, we investigate the influence of occ and occ$_C$ on the query time of locate of our indexes. For that, we focus again on ENGLISH.001.2 for patterns of fixed length $|P| = 100$, where we observe that occ$_C$ has a stronger influence on the running time than occ has.

In Figure 4.4, we study the maximum height $\tau_P = O(\lg |P|)$ that we achieved for the patterns with $|P| = 10$ to 10000 in each dataset. For this experiment, we randomly select a position j in T and extracted $P = T[j..j+9]$ to $P = T[j..j+9999]$.

For every dataset, we could observe that τ_P is logarithmic to the pattern length, especially for artificial dataset FIB41, TM29, and RS.13, where τ_P is empirically larger than measured in other datasets.

In DNA and COMMONCRAWL.ASCII.TXT, τ_P is at most 3 , but this is because $\tau_T = 3$ for these datasets.

As a practical optimization, we prematurely abort the computation of $\mathcal{G}_T$ at a certain height $\tau' < \tau_T$ when the reduction of $T^{(\tau'-1)}$ to $T^{(\tau')}$ with the newly introduced production rules would increase the grammar size (we measure the needed space in terms of GCIS-nep). Hence, we may end up with a larger right hand side of the start symbol $X^{(\tau')} \to T^{(\tau'-1)}$, which could be in fact the text itself if the text is incompressible with GCIS. This heuristic takes effect in non-highly-repetitive datasets such as DNA and COMMONCRAWL.ASCII.TXT. It also has an effect on the height τ_P since the core must have a height less than the height of the start symbol of T.

Without the heuristic, we reach a height $\tau_T = 9$ for DNA, and we also reach high values for τ_P, on average $\tau_P = 6.55$ for $|P| = 10000$, or $\tau_P = 5.81$ for $|P| = 5000$.

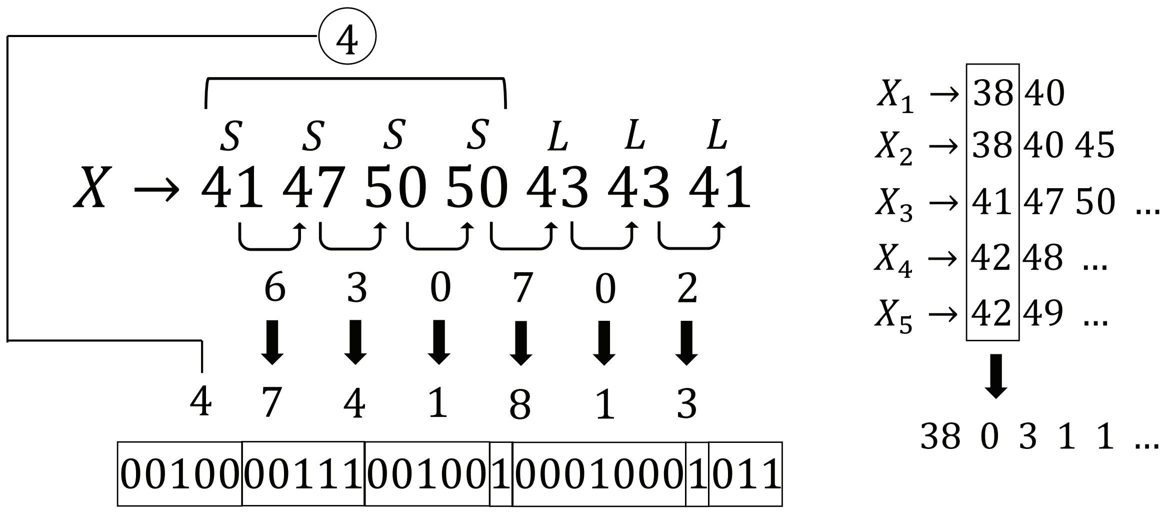


Figure 4.1: How to encode GCIS factors. The suffix type of each factor is divided two blocks, type S in left blocks and type L in right blocks. We can extract the factor from the start number (41 in this example), the differences $(6, 3, 0, 7, 0, 2)$ and the trigger that where the type is changed (4 in this example).

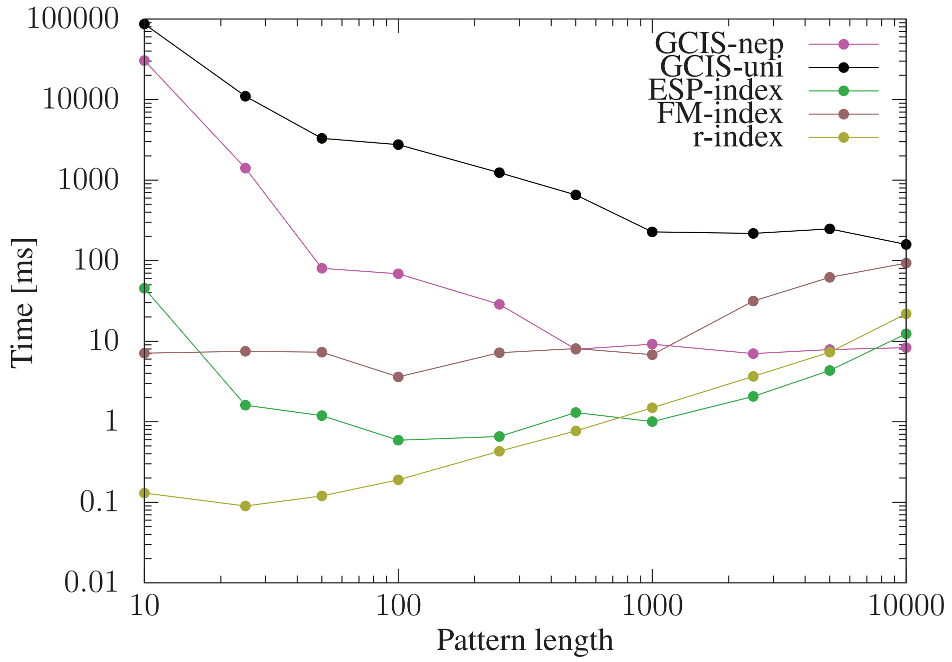


Figure 4.2: Relations of locate time and pattern length in dataset "english.001.2".

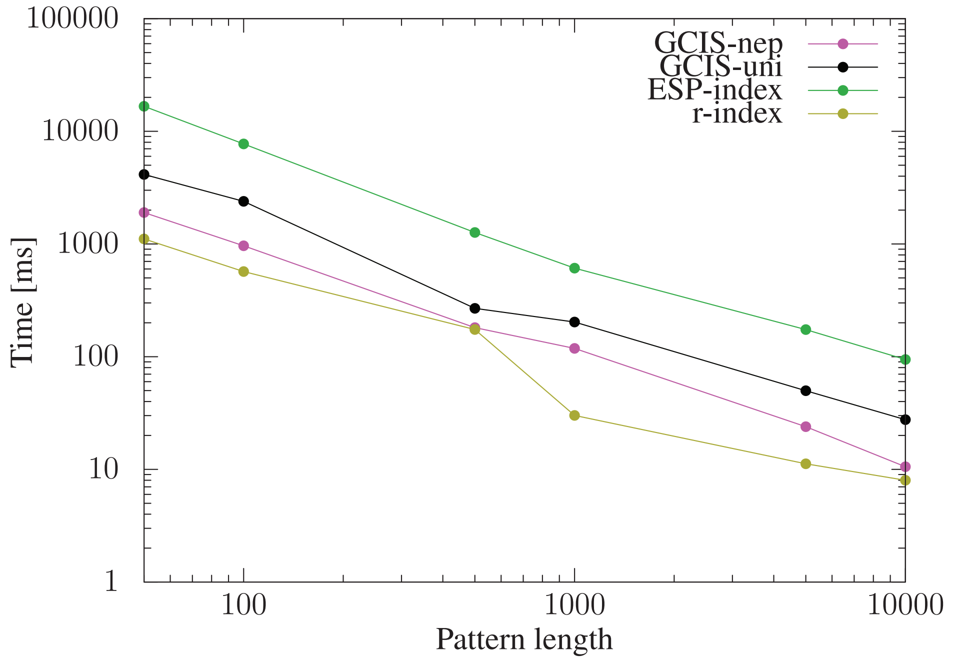


Figure 4.3: Relations of locate time and pattern length in dataset ”fib41”.

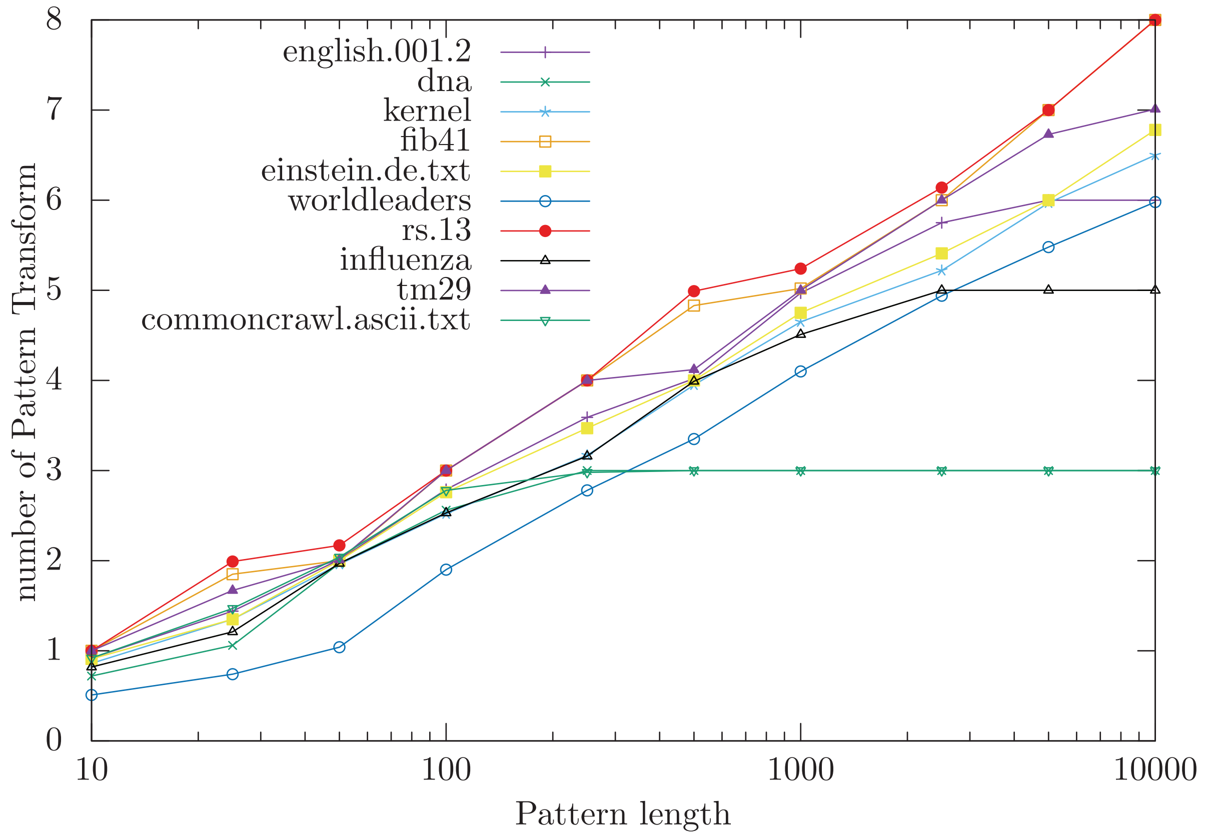


Figure 4.4: The number of transformation of random patterns of each fixed length.

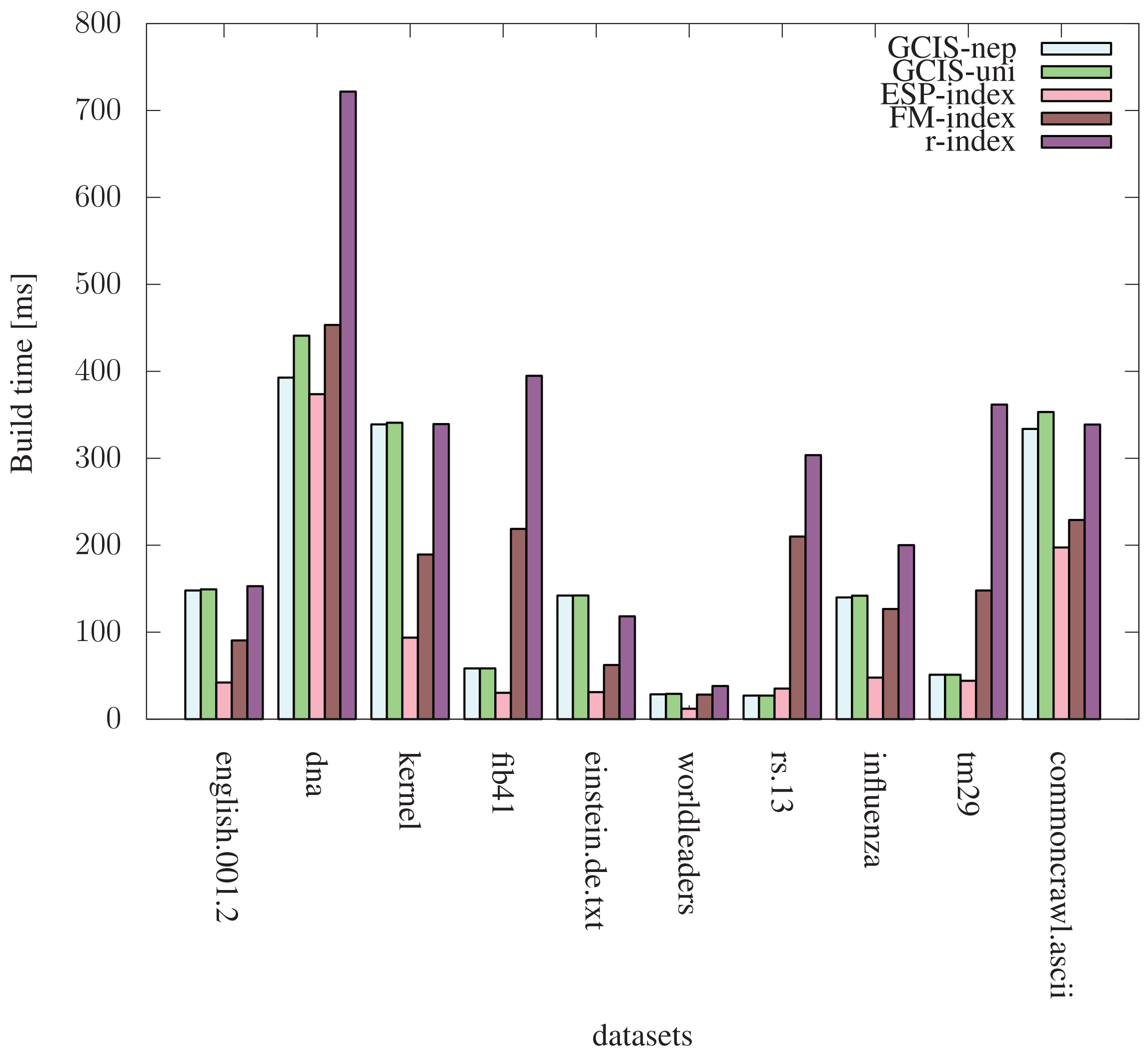


Figure 4.5: The index building time of all self-indexes used in our experiments.

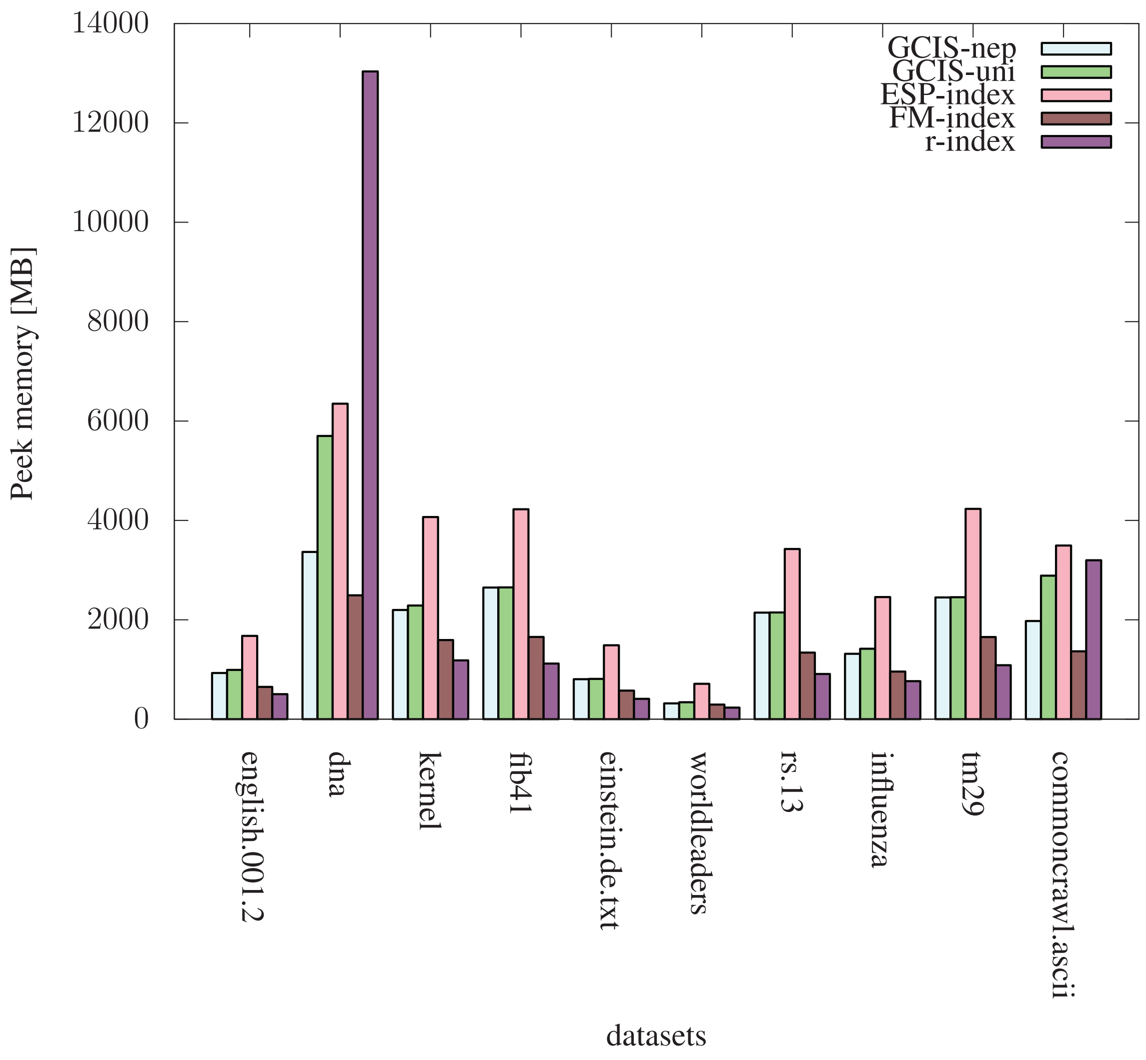


Figure 4.6: The peek memory of all self-indexes used in our experiments.

Chapter 5

Compression Sensitivity

5.1 Grammar Compression by Induced Sorting (GCIS)

In this section, we consider the worst-case sensitivity of the *grammar compression by induced sorting* (*GCIS*) [94, 93].

For convenience, we again explain the GCIS grammar construction to introduce some new symbols, which is explained in Section 5.1.1. The constructed GCIS grammar and the size is identical to Chapter 4.

5.1.1 GCIS construction

Let $i_1, \ldots, i_{z+1}$ be the sequence of the LMS positions in $T_1 = T$ sorted in increasing order. Let $D_j = T[i_j..i_{j+1} - 1]$ for any $1 \leq j \leq z$. When $z \geq 2$, then $T_1 = D_1 \ldots D_z$ is called the GCIS-parsing of T_1. Next, we create new non-terminal symbols $R_1, \ldots, R_z$ such that $R_i = 1 + \sigma + |\{D_j : D_j \prec D_i : 1 \leq j \leq z\}|$ for each i. Intuitively, we pick the least unused character from Π and assign it to R_i. Then, $T_2 = R_1 \cdots R_z$ is called the GCIS-string of T_1. Let $\mathcal{G}_1$ the set of all z symbols in T_2, and $P_1 = \{R_i \rightarrow D_i : 1 \leq i \leq z\}$ is the set of production rules. Let $\mathcal{D}_1 = \{D_1, \ldots, D_z\}$ be the set of all distinct factors. Then we define GCIS recursively, as follows:

Definition 5.1. *For $k \geq 1$, let the sequence $i_1, i_2, \ldots, i_{z_k+1}$ be all LMS positions sorted in increasing order, and $D_j = T_k[i_j \ldots i_{j+1} - 1]$ for any $1 \leq j \leq z_k$. $T_k = D_1 \ldots D_{z_k}$ is the GCIS-parsing of T_k. For all i in $1 \leq i \leq z_k$, we define R to satisfy :*

$$R_i = |\{D_j : D_j \prec D_i : 1 \leq j \leq z_k)\}| + \sum_{t=1}^{k-1} |P_t| + \sigma + 1.$$

Then, $T_{k+1} = R_1 \dots R_{z_k}$ is the GCIS-string of T_k. $\mathcal{G}_{k+1}$ is the set of non-terminals, $P_k = \{R_i \to D_i : 1 \le i \le z_k\}$ is the set of production rules. $\mathcal{D}_k = \{D_1, \dots, D_{z_k}\}$ is the set of all distinct factors in the GCIS-parsing of T_k.

Again, each R_i is chosen to be the least unused character from Π. T_{k+1} is not defined if there are no LMS positions in $T_k[2..|T_k|]$. Then, the GCIS grammar of T is $(\Sigma, \bigcup_{t=1}^{k} \mathcal{G}_t, \bigcup_{t=1}^{k-1} P_t, T_k)$. T is derived from the recursive application of the rules $\bigcup_{t=1}^{k-1} P_t$, which is the third argument, to the fourth argument T_k, which is the start string, until there are no non-terminal characters, which is in the second argument $\bigcup_{t=1}^{k} \mathcal{G}_t = \Pi$, in the string. Let $r = k$ be the height of GCIS, in other words how many times we applied this GCIS method recursively to T. Let $g_{\mathrm{is}}(T)$ be the size of GCIS grammar of T. Then, if $r = 0$, $g_{\mathrm{is}}(T) = |T|$, and if $r \ge 1$, $g_{\mathrm{is}}(T) = \|\mathcal{D}_1\| + \cdots + \|\mathcal{D}_r\| + T_r$, where $\|S\|$ for a set of strings denotes the total length of the strings in S.

Figure 5.1 shows an example on how GCIS is constructed from an input string.

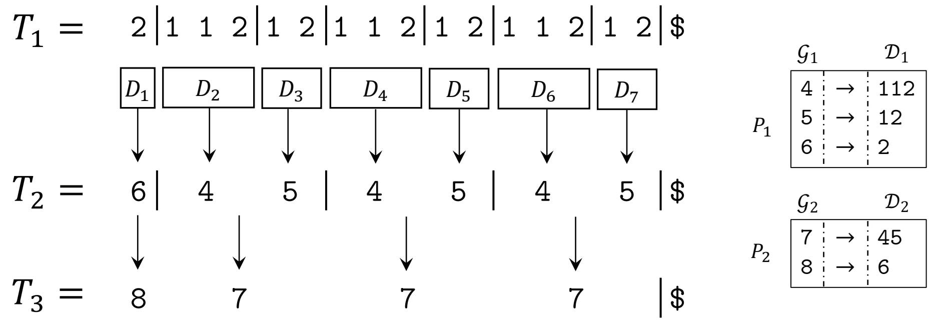

$T_1 =$ 2|1 1 2|1 2|1 1 2|1 2|1 1 2|1 2|$
D_1 D_2 D_3 D_4 D_5 D_6 D_7
$T_2 =$ 6| 4 5 | 4 5 | 4 5 |$
$T_3 =$ 8 7 7 7 |$
$\mathcal{G}_1$ $\mathcal{D}_1$
P_1
4 → 112
5 → 12
6 → 2
$\mathcal{G}_2$ $\mathcal{D}_2$
P_2
7 → 45
8 → 6

Figure 5.1: Construction of GCIS from string $T = T_1 = \mathtt{2112121121211212\$}$. In this case, there are 8 LMS positions $i_1, \dots, i_8$ in T and 7 factors $D_1, \dots, D_7$. $\mathcal{D}_1 = \{\mathtt{112}, \mathtt{12}, \mathtt{2}\}$ is the set of distinct factors of the GCIS-parsing for T_1, and $T_2 = R_1 \cdots R_7 = \mathtt{6454545}$ is the GCIS-string of T_1. Recursively, $\mathcal{D}_2 = \{\mathtt{6}, \mathtt{45}\}, T_3 = \mathtt{8777}$, and the start string of the GCIS for T_1 is T_3 because the number of factors of the GCIS-parsing of T_3 is 1 (excluding \$), in other words there are no LMS positions in $T_3[2..|T_3|]$. The size of the GCIS grammar of T is $g_{\mathrm{is}}(T) = \|\mathcal{D}_1\| + \|\mathcal{D}_2\| + |T_3| = 6 + 3 + 4 = 13$.

From now on, we consider to perform an edit operation to the input string T and will consider how the GCIS changes after the edit.

Definition 5.2. *Let S and S' be strings. If S' is obtained from S by deleting the substring of length a starting from a position c in S and by inserting a string of length b to the same position c, then we write $F(S, S') = (a, b)$.*

Our single-character edit operation performed to T can be described as $F(T,T') = (1,1)$ for substitution, $F(T,T') = (0,1)$ for insertion, and $F(T,T') = (1,0)$ for deletion. We will use this notation F to the GCIS-strings for T and T', in which case a, b can be larger than 1. Still, we will prove that a, b are small constants for the GCIS-strings.

Let $T'_1 = T'$. As with the definitions for T, $T'_k = D'_1, \ldots, D'_{z'_k}$ is the GCIS-parsing of T'_k, $T'_{k+1} = R'_1 \cdots R'_{z'_k}$ is the GCIS-string of T'_k, $\mathcal{G}'_k$ is the set of non-terminals for T'_k, $\mathcal{D}_k = \{D'_1, \ldots, D'_{z'_k}\}$ is the set of all distinct factors of the GCIS-parsing of T'_k, $P'_k = \{R_i \to D_i : 1 \leq i \leq z'_k\}$ is the set of production rules. Then we can recursively define $T'_2, T'_3 \ldots, T_{r'}$ similarly to T, where r' is the height of the GCIS for T'.

5.1.2 Upper bounds for the sensitivity of g_{is}

This section presents the following upper bounds for the sensitivity of GCIS.

Theorem 5.1. *The following upper bounds on the sensitivity of GCIS hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g_{\mathrm{is}}, n) \leq 4$. $\mathsf{AS}_{\mathrm{sub}}(g_{\mathrm{is}}, n) \leq 3g_{\mathrm{is}}$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g_{\mathrm{is}}, n) \leq 4$. $\mathsf{AS}_{\mathrm{ins}}(g_{\mathrm{is}}, n) \leq 3g_{\mathrm{is}}$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g_{\mathrm{is}}, n) \leq 4$. $\mathsf{AS}_{\mathrm{del}}(g_{\mathrm{is}}, n) \leq 3g_{\mathrm{is}}$.

We will prove this theorem as follows: We unify substitutions, insertions, and deletions by using the F function in Definition 5.2. First, we prove that edit operations do not affect the size of the GCIS grammar. Second, we divide the size of GCIS grammar $g_{\mathrm{is}}(T)$ into each $\|\mathcal{D}_k\|$ and prove that $\|\mathcal{D}'_k\| \leq 4\|\mathcal{D}_k\|$. Then, $g_{\mathrm{is}}(T') \leq 4g_{\mathrm{is}}(T)$ holds.

Lemma 5.1 ([92])**.** *The type of $T[k]$ is S if $T[k] \prec T[k+1]$ and L if $T[k] \succ T[k+1]$. If $T[k] = T[k+1]$, the type of $T[k]$ equals to the type of $T[k+1]$.*

Let $\mathrm{rank}_T[i]$ be the lexicographical rank of the character $T[i]$ at position i in T. Let $\hat{T}$ be any string of length $|\hat{T}| = |T|$ such that $\mathrm{rank}_{\hat{T}}[i] = \mathrm{rank}_T[i]$ for every $1 \leq i \leq |T|$.

Definition 5.3. *If $\hat{T}$ is the string that can be obtained by replacing the characters in T without changing the ranks of any characters in T, we write $F^*(T, \hat{T}) = (0,0)$.*

Lemma 5.2. *If $F^*(T, \hat{T}) = 0$, then $g_{\mathrm{is}}(\hat{T}) = g_{\mathrm{is}}(T)$.*

Proof. The lemma immediately follows from Lemma 5.1 and definition 5.3 and that $\mathrm{rank}_{\hat{T}}[i] = \mathrm{rank}_T[i]$ for every $1 \leq i \leq |T|$. □

Definition 5.4. *If $F^*(T,\hat{T}) = (0,0)$, $F(\hat{T},\hat{T}') = (a,b)$, and $F^*(\hat{T}',T') = (0,0)$, we write $F^*(T,T') = (a,b)$.*

Figure 5.2 shows a concrete example for Lemma 5.2.

```
T =      2|1 2 3|1 3|1 2 3|1 3|1 2 3|1 3|$
    G_1 =    6|4 5|4 5|4 5|$

T̂ =      2|1 4 5|1 5|1 4 5|1 5|1 4 5|1 5|$
    Ĝ_1 =    8|6 7|6 7|6 7|$
```

Figure 5.2: Two strings T and $\hat{T}$ that can be obtained by replacing some characters in T without changing the relative order of any characters, result in the same number of factors in the GCIS-parsing, and each length exactly matches in both of the strings. Therefore, $\|\mathcal{D}_1\| = \|\hat{\mathcal{D}}_1\|$ and $|T_2| = |\hat{T}_2|$ holds, and $\hat{T}_2$ is recursively the string that can be obtained by replacing some characters in T_2 without changing the relative order of any characters. Therefore, we can consider the size of GCIS of T using such a string $\hat{T}$ instead of T itself.

A natural consequence of Lemma 5.2 is that edit operations which do not change the relative order of the characters in T do not affect the size of the grammar.

From now on, we analyze how the size of the GCIS of the string T can increase after the edit operation in the string T'. In the following lemmas, let $1 \leq h \leq r$, where r is the height of the GCIS grammar for T.

Lemma 5.3. *If $F(T_h, T'_h) = (x,y)$, then $|\mathcal{D}_h \setminus \mathcal{D}'_h| \leq 2 + \lceil x/2 \rceil$.*

Proof. First, let c be the position such that T'_h can be obtained from T_h by deleting a substring of length x from position c and inserting a substring of length y to position c. Let z and z' be the numbers of factors in the GCIS-parsing of T_h and T'_h, respectively. See Figure 5.3. Considering k where $i_k \leq c < i_{k+1}$ in T_h, the LMS positions $i_1, \ldots, i_{k-1}$ are also the LMS positions in T'_h, and for all j where $1 \leq j \leq k-2$, $D_j = D'_j$ holds. Similarly, for l where $i_{z-l-1} \leq c + x < i_{z-l}$ in T_h, the positions $i_{z-l}, \ldots, i_z$ and corresponding positions $i_{z'-l}, \ldots, i_{z'}$ in T'_h are also LMS positions. Therefore, for $z - l \leq j \leq z$ and $j' = j + (z' - z)$, $D_j = D'_{j'}$. Note that $i_{z-l-1} - i_k < x$. Since $|D_j| \geq 2$, $|D'_j| \geq 2$ with $2 \leq j \leq z$, we obtain $|\mathcal{D}_h \setminus \mathcal{D}'_h| \leq |\{D_{k-1} \ldots D_{z-l-1}\}| \leq 2 + \lceil x/2 \rceil$. □

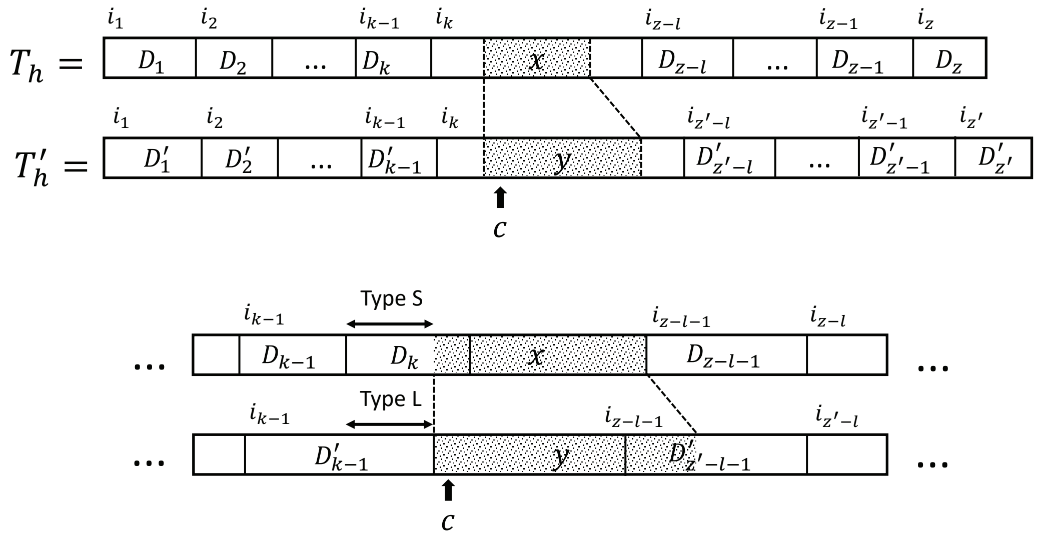

i_1 i_2 i_{k-1} i_k i_{z-l} i_{z-1} i_z
$T_h =$ D_1 D_2 … D_k x D_{z-l} … D_{z-1} D_z
i_1 i_2 i_{k-1} i_k $i_{z'-l}$ $i_{z'-1}$ $i_{z'}$
$T'_h =$ D'_1 D'_2 … D'_{k-1} y $D'_{z'-l}$ … $D'_{z'-1}$ $D'_{z'}$
c
Type S
i_{k-1} i_{z-l-1} i_{z-l}
D_{k-1} D_k x D_{z-l-1}
Type L
i_{k-1} i_{z-l-1} $i_{z'-l}$
D'_{k-1} y $D'_{z'-l-1}$
c

Figure 5.3: The case of $F(T_h, T'_h) = (x, y)$. When a substring of length x is replaced by an another string of y at position c, most of parsing are not changed. The under example show the detail of (the worst case of) the around of edited position. We can see $D_{k-1}, ..., D_{z-l-1}$ could be changed to some new blocks, and no other block never changes.

Lemma 5.4. *If $F(T_h, T'_h) = (x, y)$, then $|T'_{h+1}| \leq |T_{h+1}| + 1 + \lfloor y/2 \rfloor$.*

Proof. Let c be the position such that T'_h can be obtained from T_h by deleting a substring of length x from position c and inserting a substring of length y to position c. If $c = 1$, $|T'_{h+1}| \leq |T_{h+1}| + \lceil y/2 \rceil$ holds since there are at most y positions which can be a new LMS position in T'_h.

Assume $c \geq 2$. By Lemma 5.3, most of LMS position of T'_h corresponds to T_h. Let $c' < c$ be the right most position such that $T[c'] \neq T[c-1]$. LMS positions i with $c'+1 \leq i \leq c+x$ could be lost, and the others have corresponding position in T'_h. LMS positions i with $c'+1 \leq i \leq c+y$ could be generated, and the others have corresponding position in T_h. Note that any LMS position are not adjacent except position 1. Moreover, $c' + 1$ and c cannot be LMS positions at the same time, any position $c' + 2, \cdot, c - 1$ cannot be LMS position. Then, the increase of LMS positions is at most $1 + \lceil y/2 \rceil$. Additionally, if both position $c' + 1$ and $c + y$ is a new LMS position in T'_h, there is at least an LMS position in $T_h[c' + 1..c + x]$ (See Figure 5.4). Finally, the increase of LMS positions is at most $1 + \lfloor y/2 \rfloor$. Therefore, the increase of the factor of T'_h is $1 + \lfloor y/2 \rfloor$. (Note that at least a factor of T_h is lost.) $\square$

Lemma 5.5. *Assume $F(T_h, T'_h) = (x, y)$ and $x \leq 4, y \leq 4$. Then $F^*(T_{h+1}, T'_{h+1}) = (x', y')$, where $x' \leq 4, y' \leq 4$.*

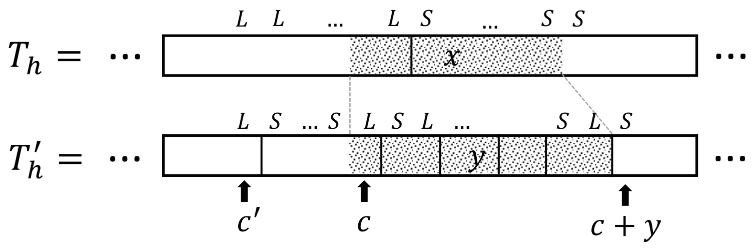


Figure 5.4: The case that there are $1 + \lceil y/2 \rceil$ new LMS positions in T'. The candidates of new LMS position in T'_h is $c' + 1, c, c + 1, \ldots, c + y$. These $y + 2$ position LMS position only can choose position that are non-adjacent, which leads to $\lceil (y + 2)/2 \rceil = 1 + \lceil y/2 \rceil$. Therefore, T'_h divide a factor of T_h into at most $2 + \lceil y/2 \rceil$ new factors. In addition, the position $c' + 1$ in T_h must be of type L to turn position $c' + 1$ to a new LMS position in T'_h. The position $c + x$ in T_h must be of type S to turn position $c + y$ to a new LMS position in T'_h. Then, there is at least an LMS position in $T[c' + 1..c + x]$.

Proof. Assume $T_h = D_1, \ldots, D_z$ and $T'_h = D'_1, \ldots, D'_{z'}$ are the GCIS-parsings of T_h and T'_h, respectively. By Lemma 5.3, there are at most $j = 2+\lceil x/2 \rceil$ consecutive factors $D_i, \ldots, D_{i+j-1}$ in $\mathcal{D}_{h+1} \setminus \mathcal{D}'_{h+1}$, and at most $\hat{j} = 2+\lceil y/2 \rceil$ consecutive factors $D_{\hat{i}}, \ldots, D_{\hat{i}+\hat{j}-1}$ in $\mathcal{D}'_{h+1} \setminus \mathcal{D}_{h+1}$ and $D_k = D'_k$ for all $1 \le k \le \max(i, \hat{i})$, and $D_{z-k} = D'_{z'-k}$ for all $0 \le k \le \max(z - i - j - 1, z' - \hat{i} - \hat{j} - 1)$. By Lemma 5.4, $j + \hat{j} \le 4 + \lceil y/2 \rceil + \lceil x/2 \rceil$. Let

$$\begin{aligned} \hat{S}_p &= |\{D_s : D_s \prec D_p (1 \le s \le z)\}| + |\{D'_s : D'_s \prec D_p (1 \le s \le z')\}|, \\ \hat{S}'_p &= |\{D_s : D_s \prec D'_p (1 \le s \le z)\}| + |\{D'_s : D'_s \prec D'_p (1 \le s \le z')\}|. \end{aligned}$$

Then, the string $\hat{T}_{h+1} = \hat{S}_1 \cdots \hat{S}_z$ can be obtained from T_{h+1} by replacing some characters in T_{h+1}, T'_{h+1} without changing the relative order of any characters, and $\hat{T'_{h+1}}$ as well. In addition, $F(\hat{T}_{h+1}, \hat{T}'_{h+1}) = (j, \hat{j})$ holds because $\hat{R}_k = \hat{R}'_k$ for all $1 \le k \le \max(i, \hat{i})$, and $\hat{R}_{z-k} = \hat{R}'_{z'-k}$ for all $0 \le k \le \max(z - i - j - 1, z' - \hat{i} - \hat{j} - 1)$, which leads to $F^*(T_h, T'_h) = (j, \hat{j})$. See Figure 5.5. □

Lemma 5.6. *If* $F(T, T') \in \{(1, 1), (1, 0), (0, 1)\}$, *then* $F^*(\hat{T}_h, \hat{T}'_h) = (x, y)$, *where* $x \le 4, y \le 4, h \le r$.

Proof. Immediately follows from Lemma 5.5. □

Lemma 5.7. *If* $F(T_h, T'_h) = (x, y)$, $\|\mathcal{D}'_h\| \le 4\|\mathcal{D}_h\| - 1 + y$.

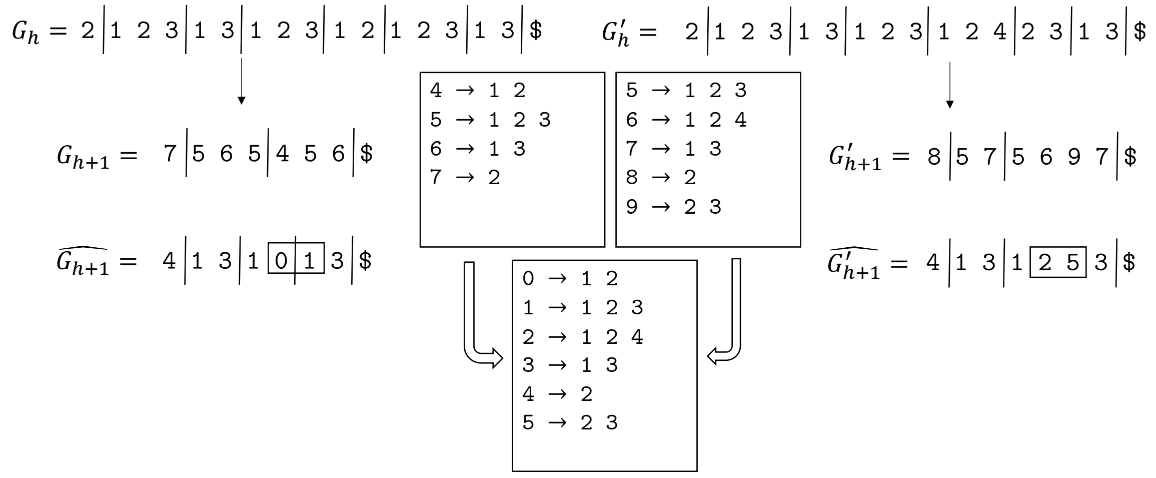

G_h = 2|1 2 3|1 3|1 2 3|1 2|1 2 3|1 3|$
G'_h = 2|1 2 3|1 3|1 2 3|1 2 4|2 3|1 3|$
G_{h+1} = 7|5 6 5|4 5 6|$
4 → 1 2
5 → 1 2 3
6 → 1 3
7 → 2
5 → 1 2 3
6 → 1 2 4
7 → 1 3
8 → 2
9 → 2 3
G'_{h+1} = 8|5 7|5 6 9 7|$
$\widehat{G_{h+1}}$ = 4|1 3|1 0 1 3|$
0 → 1 2
1 → 1 2 3
2 → 1 2 4
3 → 1 3
4 → 2
5 → 2 3
$\widehat{G'_{h+1}}$ = 4|1 3|1 2 5 3|$

Figure 5.5: Examples of $\hat{T_{h+1}}$ and $\hat{T}'_{h+1}$ for strings T_h and T'_h, where T'_h can be obtained from T_h by substituting a 1 with a 4. The box under the 2 other boxes shows the "common" productions common to $\hat{T_{h+1}}$ and $\hat{T'_{h+1}}$, e.g., by applying $1 \to 123$ to the occurrences of 1 in $\hat{T}_{h+1}$ and $\hat{T}'_{h+1}$, we obtain the corresponding substrings 123 in T_h and T'_h. Since the common number is assigned to each equal factors in T_h and T'_h, the size of the symmetric difference of $\hat{T}_{h+1}$ and $\hat{T}'_{h+1}$ equals to the number of factors changed by substitution, insertion, or deletion from T_h to T'_h, which is four in this example.

Proof. Considering k where $i_k \leq c < i_{k+1}$ in T_h and l where $i_{z-l-1} \leq c+x < i_{z-l}$. As you can see Figure 5.3 and from lemma 5.3, $\|\mathcal{D}'_h\|$ grow up by at most $|D_{k-1}| + |D_k| + \ldots + |D_{z-l-1}|$. However, most of the factors overlap x, therefore $\|\mathcal{D}'_h\|$ only grow up by at most $|D_{k-1}| + |D_k| + |D_{z-l-1}| + (y-x)$. Even if the 3 factors are the same and $\mathcal{D}'_h$ consists of only the factor, The total length of new factors to be added in $\mathcal{D}'_h$ is at most $4\|\mathcal{D}_h\| + (y-x)$.

If $x \geq 1$, $\|\mathcal{D}'_h\| \leq 4\|\mathcal{D}_h\| - 1 + y$. If $x = 0$, at most 2 factors of $\mathcal{D}'_h$ can changed by Lemma 5.3, We show $\|\mathcal{D}'_h\| \leq 3\|\mathcal{D}_h\| + y \leq 4\|\mathcal{D}_h\| - 1 + y$. □

Lemma 5.8. *Assume $F(T_h, T'_h) = (x, y)$ with $y \leq 4, h < r$. Then, $\|\mathcal{D}'_h\| \leq 4\|\mathcal{D}_h\|$.*

Proof. Since $h < r$, there are at least 2 rules of at least length 2 in P_h. By Lemma 5.7, The number of factors not included in the editing range is at most 3. Even if the 3 factors are the length of $\|\mathcal{D}_h\| - 2$, The total length of new factors to be added in $\mathcal{D}'_h$, is at most $4\|\mathcal{D}_h\| - 6 + (y-x) \leq 4\|\mathcal{D}_h\|$. □

Lemma 5.9. *Assume $F(T_{r-1}, T'_{r-1}) = (x, y)$ with $y \leq 4$. Then, $g_{\mathrm{is}}(T'_{r-1}) = \|\mathcal{D}'_{r-1}\| + g_{\mathrm{is}}(T'_r) \leq 4\|\mathcal{D}_{r-1}\| + 4|T_r| + y$.*

Proof. Consider the special case of lemma 5.8. Even if all factors of T_{r-1} changed, $\mathcal{D'}_{r-1} \leq |T_r|\|\mathcal{D}_{r-1}\| + 4$. Since $g_{\text{is}}(T) \leq 2|T|$ by GCIS construction, If $|T_r| \geq 4$, $g_{\text{is}}(T'_r) \leq 2|T'_r| \leq 4|T_r| = 4g_{\text{is}}(T_r)$ holds by Lemma 5.5 and 5.4. If $|T_{h+1}| \leq 3$, let $p = 4 - |T_{h+1}|$. Then $\mathcal{D'}_{r-1} \leq |T_r|\|\mathcal{D}_{r-1}\| + y \leq 4\|\mathcal{D}_{r-1}\| - 2p + y$, and $g_{\text{is}}(T'_r) \leq 2|T'_r| \leq 2(2|T_r| + p) \leq 4g_{\text{is}}(T) + 2p$. Finally, $\mathcal{D'}_{r-1} + g_{\text{is}}(T'_r) \leq 4\|\mathcal{D}_{r-1}\| + 4|T_r| + y$. □

Lemma 5.10. *If $F(T_1, T'_1) \in \{(1,1),(1,0),(0,1)\}$, then $g_{\text{is}}(T'_1) \leq 4g_{\text{is}}(T_1)$.*

Proof. Assume $F(T_1, T'_1) \in \{(1,1),(1,0),(0,1)\}$. If $r = 2$, then $\|\mathcal{D'}_1\| \leq 4\|\mathcal{D}_1\|$ holds from Lemma 5.7. Since $|T'_2| \leq |T_2| + 1$ by Lemma 5.4, $g_{\text{is}}(T'_2) \leq 2|T'_2| \leq 2|T_2| + 2 = 4g_{\text{is}}(T_2)$ holds.

Assume $r > 2$. Then, $\|\mathcal{D'}_1\| \leq 4\|\mathcal{D}_1\| - 5$ holds by Lemma 5.8. By application of Lemma 5.8 and 5.9, we obtain,

$$\begin{aligned}
g_{\text{is}}(T'_1) &= ||\mathcal{D'}_1|| + \ldots + ||\mathcal{D'}_{r-1}|| + g_{\text{is}}(T'_r) \\
&\leq (4||\mathcal{D}_1|| - 5) + 4||\mathcal{D}_2|| + 4||\mathcal{D}_{r-2}|| + g_{\text{is}}(T'_{r-1}) \\
&\leq (4||\mathcal{D}_1|| - 5) + 4||\mathcal{D}_2|| + 4||\mathcal{D}_{r-1}|| + (4\|\mathcal{D}_{r-1}\| + 4|T_r| + 4) \\
&\leq 4||\mathcal{D}_1|| + 4||\mathcal{D}_2|| + 4||\mathcal{D}_{r-1}|| + 4\|\mathcal{D}_{r-1}\| + 4|T_r| \\
&\leq 4g_{\text{is}}(T_1).
\end{aligned}$$

□

5.1.3 Lower bounds for the sensitivity of g_{is}

Theorem 5.2. *The following lower bounds on the sensitivity of GCIS hold:*
substitutions: $\liminf_{n\to\infty} \mathsf{MS}_{\text{sub}}(g_{\text{is}}, n) \geq 4$. $\mathsf{AS}_{\text{sub}}(g_{\text{is}}, n) \geq 3g_{\text{is}} - 13 \in \Omega(n)$.
insertions: $\liminf_{n\to\infty} \mathsf{MS}_{\text{ins}}(g_{\text{is}}, n) \geq 4$. $\mathsf{AS}_{\text{ins}}(g_{\text{is}}, n) \geq 3g_{\text{is}} - 22 \in \Omega(n)$.
deletions: $\liminf_{n\to\infty} \mathsf{MS}_{\text{del}}(g_{\text{is}}, n) \geq 4$. $\mathsf{AS}_{\text{del}}(g_{\text{is}}, n) \geq 3g_{\text{is}} - 26 \in \Omega(n)$.

Proof. Assume $p > 1$.

substitutions: Consider the following string of length $n = 4p + 4 \in \Theta(p)$:

$$T = \mathtt{2}^p\mathtt{3}\mathtt{2}^p\mathtt{3}\mathtt{2}^p\mathtt{3}\mathtt{2}^p\mathtt{3}$$

By the construction of the GCIS grammar of T, we obtain $\mathcal{D}_1 = \{\mathtt{2}^p\mathtt{3}\}$, $T_2 = \mathtt{4444}$, $g_{\text{is}}(T) = \|\mathcal{D}_1\| + |T_2| = p + 5$. The following string

$$T' = \mathtt{2}^p\mathtt{3}\mathtt{2}^p\mathtt{3}\mathtt{2}^p\mathtt{1}\mathtt{2}^p\mathtt{3}$$

can be obtained from T by substituting the third $\mathtt{3}$ with $\mathtt{1}$. By the construction of GCIS grammar of T, we obtain $T_2' = \mathtt{564}, \mathcal{D}'_1 = \{\mathtt{12}^p\mathtt{3}, \mathtt{2}^p\mathtt{3}, \mathtt{2}^p\mathtt{32}^p\}$, $g_{\text{is}}(T') = \|\mathcal{D}'_1\| + |T_2'| = 4p + 7$, which leads to $\liminf_{n\to\infty} \mathsf{MS}_{\text{sub}}(g_{\text{is}}, n) \geq \liminf_{p\to\infty}(4p+7)/(p+5) = 4$, and $\mathsf{AS}_{\text{sub}}(g_{\text{is}}, n) = (4p+7) - (p+5) = 3p+2 = 3g_{\text{is}} - 13 \in \Omega(n)$.

insertions: Consider the following string of length $n = 8p + 8 \in \Theta(p)$:

$$T = (\mathtt{12})^p\mathtt{13}(\mathtt{12})^p\mathtt{13}(\mathtt{12})^p\mathtt{13}(\mathtt{12})^p\mathtt{13}$$

By the construction of the GCIS grammar, we obtain $\mathcal{D}_1 = \{\mathtt{12}, \mathtt{13}\}$, $T_2 = \mathtt{3}^p\mathtt{43}^p\mathtt{43}^p\mathtt{43}^p\mathtt{4}$, $\mathcal{D}_2 = \{\mathtt{3}^p\mathtt{4}\}$, $T_3 = \mathtt{5555}$, $g_{\text{is}}(T) = \|\mathcal{D}_1\| + \|\mathcal{D}_2\| + |T_3| = 4 + (p+1) + 4 = p + 9$.

The string

$$T' = (\mathtt{12})^p\mathtt{13}(\mathtt{12})^p\mathtt{13}(\mathtt{12})^p\mathtt{113}(\mathtt{12})^p\mathtt{13}$$

can be obtained from T by inserting $\mathtt{1}$ to just before the third $(\mathtt{12})^p\mathtt{1}$. By the construction of GCIS grammar of T', we obtain $\mathcal{D}_1 = \{\mathtt{12}, \mathtt{13}, \mathtt{113}\}, T_2' = \mathtt{5}^p\mathtt{65}^p\mathtt{65}^p\mathtt{45}^p\mathtt{6}, T_3' = \mathtt{897}, \mathcal{D}'_2 = \{\mathtt{45}^p\mathtt{6}, \mathtt{5}^p\mathtt{6}, \mathtt{5}^p\mathtt{65}^p\}$, $g_{\text{is}}(T') = \|\mathcal{D}'_1\| + \|\mathcal{D}'_2\| + |T_3'| = 7 + (4p+4) + 3 = 4p + 14$, which leads to $\liminf_{n\to\infty} \mathsf{MS}_{\text{ins}}(g_{\text{is}}, n) \geq \liminf_{p\to\infty}(4p+14)/(p+9) = 4$, and $\mathsf{AS}_{\text{ins}}(g_{\text{is}}, n) = (4p+14) - (p+9) = 3p+5 = 3g_{\text{is}} - 22 \in \Omega(n)$.

deletions: Consider the following string of length $n = 8p + 12 \in \Theta(p)$:

$$T = (\mathtt{13})^p\mathtt{132}(\mathtt{13})^p\mathtt{132}(\mathtt{13})^p\mathtt{132}(\mathtt{13})^p\mathtt{132}$$

By the construction of the GCIS grammar of T, we obtain $\mathcal{D}_1 = \{\mathtt{13}, \mathtt{132}\}, T_2 = \mathtt{4}^p\mathtt{54}^p\mathtt{54}^p\mathtt{54}^p\mathtt{5}$, $\mathcal{D}_2 = \{\mathtt{4}^p\mathtt{5}\}$, $T_3 = \mathtt{6666}$, $g_{\text{is}}(T) = \|\mathcal{D}_1\| + \|\mathcal{D}_2\| + |T_3| = 5 + (p+1) + 4 = p + 10$.

The string

$$T' = (\mathtt{13})^p\mathtt{132}(\mathtt{13})^p\mathtt{132}(\mathtt{13})^p\mathtt{12}(\mathtt{13})^p\mathtt{132}$$

can be obtained from T by deleting $\mathtt{3}$ in the third $\mathtt{132}$. By the construction of GCIS grammar of T', we obtain $\mathcal{D}_1 = \{\mathtt{12}, \mathtt{13}, \mathtt{132}\}, T_2' = \mathtt{5}^p\mathtt{65}^p\mathtt{65}^p\mathtt{45}^p\mathtt{6}, \mathcal{D}'_2 = \{\mathtt{45}^p\mathtt{6}, \mathtt{5}^p\mathtt{6}, \mathtt{5}^p\mathtt{65}^p\}, T_3' = \mathtt{897}, g_{\text{is}}(T') = \|\mathcal{D}'_1\| + \|\mathcal{D}'_2\| + |T_3'| = 7 + (4p+4) + 3 = 4p + 14$, which leads to $\liminf_{n\to\infty} \mathsf{MS}_{\text{del}}(g_{\text{is}}, n) \geq \liminf_{p\to\infty}(4p+14)/(p+10) = 4$, and $\mathsf{AS}_{\text{del}}(g_{\text{is}}, n) = (4p+14) - (p+10) = 3p+4 = 3g_{\text{is}} - 26 \in \Omega(n)$. □

5.2 Bisection

In this section, we consider the worst-case sensitivity of the compression algorithm *bisection* [87] which is a kind of grammar-based compression that has a tight connection to BDDs.

Given a string T of length n, the bisection algorithm builds a grammar generating T as follows. We consider a binary tree $\mathcal{T}$ whose root corresponds to T. The left and right children of the root correspond to $T_1 = T[1..2^j]$ and $T_2 = T[2^j+1..n]$, respectively, where j is the largest integer such that $2^j < n$. We apply the same rule to T_1 and to T_2 recursively, until obtaining single characters which are the leaves of $\mathcal{T}$. After $\mathcal{T}$ is built, we assign a label (non-terminal) to each node of $\mathcal{T}$. If there are multiple nodes such that the leaves of their subtrees are the same substrings of T, we label the same non-terminal to all these nodes. The labeled tree $\mathcal{T}$ is the derivation tree of the bisection grammar for T. We denote by $g_{\mathrm{bsc}}(T)$ the size of the bisection grammar for T.

5.2.1 Lower bounds for the sensitivity of g_{bsc}

Theorem 5.3. *The following lower bounds on the sensitivity of g_{bsc} hold:*
substitutions: $\liminf_{n\to\infty} \mathsf{MS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \geq 2$. $\mathsf{AS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \geq g_{\mathrm{bsc}} - 4$ *and* $\mathsf{AS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \geq 2\log_2 n - 4$.
insertions: $\liminf_{n\to\infty} \mathsf{MS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \geq \sigma$. $\mathsf{AS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \in \Omega(\sigma g_{\mathrm{bsc}})$ *and* $\mathsf{AS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \in \Omega\left(\sigma^2\left(\log\frac{n}{\sigma}\right)\right)$.
deletions: $\liminf_{n\to\infty} \mathsf{MS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \geq \sigma$. $\mathsf{AS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \in \Omega(\sigma g_{\mathrm{bsc}})$ *and* $\mathsf{AS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \in \Omega\left(\sigma^2\left(\log\frac{n}{\sigma}\right)\right)$.

Proof. **substitutions:** Consider a unary string $T = \mathtt{a}^n$ with $n = 2^k$. The set of productions for T is

$$\begin{aligned}
X_1 &= \mathtt{a} && \text{(generating a)},\\
X_2 &= X_1X_1 && \text{(generating aa)},\\
X_3 &= X_2X_2 && \text{(generating aaaa)},\\
&\dots\\
X_k &= X_{k-1}X_{k-1} && (\text{generating } \mathtt{a}^{2^k}),
\end{aligned}$$

with $g_{\mathrm{bsc}}(T) = 2k-1$. Let $T' = \mathtt{a}^{n-1}\mathtt{b}$ that can be obtained by replacing the last $\mathtt{a}$ in T with $\mathtt{b}$.

The set of productions for T' is

$$
\begin{array}{ll}
X_1 = \mathtt{a} & \text{(generating } \mathtt{a}\text{)},\\
X_2 = X_1X_1 & \text{(generating } \mathtt{aa}\text{)},\\
X_3 = X_2X_2 & \text{(generating } \mathtt{aaaa}\text{)},\\
\dots & \\
X_{k-1} = X_{k-2}X_{k-2} & \text{(generating } \mathtt{a}^{2^{k-1}}\text{)},\\
Y_1 = \mathtt{b} & \text{(generating } \mathtt{b}\text{)},\\
Y_2 = X_1Y_1 & \text{(generating } \mathtt{ab}\text{)},\\
Y_3 = X_2Y_2 & \text{(generating } \mathtt{aaab}\text{)},\\
\dots & \\
Y_k = X_{k-1}Y_{k-1} & \text{(generating } \mathtt{a}^{2^k-1}\mathtt{b}\text{)}
\end{array}
$$

with $g_{\text{bsc}}(T') = 2k - 1 + 2(k-1) - 1 = 4k - 4$. Thus $\liminf_{n\to\infty} \mathsf{MS}_{\text{sub}}(g_{\text{bsc}}, n) \geq \liminf_{k\to\infty} \frac{4k-4}{2k-1} \geq 2$. Also, $\mathsf{AS}_{\text{sub}}(g_{\text{bsc}}, n) \geq (4k-4) - (2k-1) = 2k - 5 = g_{\text{bsc}}(T) - 4$ and $\mathsf{AS}_{\text{sub}}(g_{\text{bsc}}, n) \geq 2\log_2 n - 4$ as $k = \log_2 n$.

deletions: Assume that $\sigma = 2^i$ with a positive integer $i \geq 1$. Let Q be a string that contains $t = \sigma^2$ distinct bigrams and $|Q| = \sigma^2 + 1$. Let $Q' = Q[2..|Q|]$. We consider the string

$$T \quad = \quad Q'[1]^{2^p} \cdots Q'[|Q'|]^{2^p}.$$

Note that $p = \log(n/\sigma)$. The set of productions for T from depth 1 to p is:

$$
\begin{array}{rcl}
X_i & \rightarrow & \sigma_i\sigma_i \quad (1 \leq i \leq p),\\
X_{p\sigma+i} & \rightarrow & X_{(p-1)\sigma+i}X_{(p-1)\sigma+i} \quad (1 \leq i \leq \sigma, 2 \leq k \leq p).
\end{array}
$$

Thus, the derivation tree $\mathcal{T}$ has $p\sigma$ internal nodes with distinct labels. Additionally, after height σ, the string consists of $t-1$ distinct bigrams, and there is no run of length 2. Then the derivation tree $\mathcal{T}$ has $t-1$ internal nodes with distinct labels in height above p. Finally, $g_{\text{bsc}}(T) = p\sigma + t - 1$.

We consider the string T' where $T[1]$ is removed, namely,

$$T' \quad = \quad T[2..|T|] = Q'[1]^{2^p-1}Q'[2]^{2^p} \cdots Q'[|Q'|]^{2^p}.$$

The set of productions for T' of height 1 is:

$$X_{(i-1)\sigma+j} \quad \rightarrow \quad \sigma_i\sigma_j \quad (1 \leq i \leq \sigma, 1 \leq j \leq \sigma).$$

Thus, the derivation tree $\mathcal{T}'$ for string T' has $t = \sigma^2$ internal nodes with distinct labels at height one. Because of this, the number of internal nodes of the derivation tree $\mathcal{T}'$ in each height $2 \leq p' \leq p$ is also at least $t = \sigma^2$. After that, the string of height p consists of t distinct bigrams, and there is no run of length 2, which is the same condition of T. Then the derivation tree $\mathcal{T}$ has additional $t-1$ internal nodes with distinct labels in height above p. Finally, $g_{\mathrm{bsc}}(T') = tp + t$. Then, we obtain:

$$\begin{aligned}
\mathsf{MS}_{\mathrm{ins}}(g_{bsc}, n) &\geq \lim_{n\to\infty} \frac{tp+t}{p\sigma + t - 1} = \frac{t}{\sigma} \geq \sigma, \\
\mathsf{AS}_{\mathrm{ins}}(g_{bsc}, n) &\geq \lim_{n\to\infty} (tp+t) - (p\sigma + t - 1) \in \Omega(\sigma \cdot p\sigma) = \Omega(\sigma g_{bsc}).
\end{aligned}$$

insertions: We use the same string T as in the case of deletions. We consider the string T' that is obtained by prepending $Q[1]$ to T, namely,

$$T' = Q[1]T = Q[1]Q'[1]^{2^p} \cdots Q'[|Q'|]^{2^p}.$$

The set of productions for T' of height 1 is:

$$\begin{aligned}
X_{(i-1)\sigma+j} &\rightarrow \sigma_i\sigma_j. \quad (1 \leq i \leq \sigma, 1 \leq j \leq \sigma) \\
X_{\sigma^2+1} &\rightarrow Q[1].
\end{aligned}$$

Thus, the derivation tree $\mathcal{T}'$ has $t+1$ internal nodes with distinct labels at height one. Because of this, the number of internal nodes of derivation tree $\mathcal{T}'$ of each height $2 \leq p' \leq p$ is also at least $t = \sigma^2$ nodes. After that, the string of height p consists of t distinct bigrams, and there is no run of length 2, which is the same condition of T. Then derivation tree $\mathcal{T}$ has additional $t-1$ internal nodes with distinct labels in height above p. Finally, $g_{\mathrm{bsc}}(T') = (t+1)p + t$. Then, we obtain:

$$\begin{aligned}
\mathsf{MS}_{\mathrm{ins}}(g_{bsc}, n) &\geq \lim_{n\to\infty} \frac{(t+1)p+t}{p\sigma + t - 1} = \frac{(t+1)}{\sigma} \geq \sigma, \\
\mathsf{AS}_{\mathrm{ins}}(g_{bsc}, n) &\geq \lim_{n\to\infty} ((t+1)p+t) - (p\sigma + t - 1) \in \Omega(\sigma \cdot p\sigma) = \Omega(\sigma g_{bsc}).
\end{aligned}$$

□

We show a concrete example of how the derivation tree changes by an insertion in Figure 5.6.

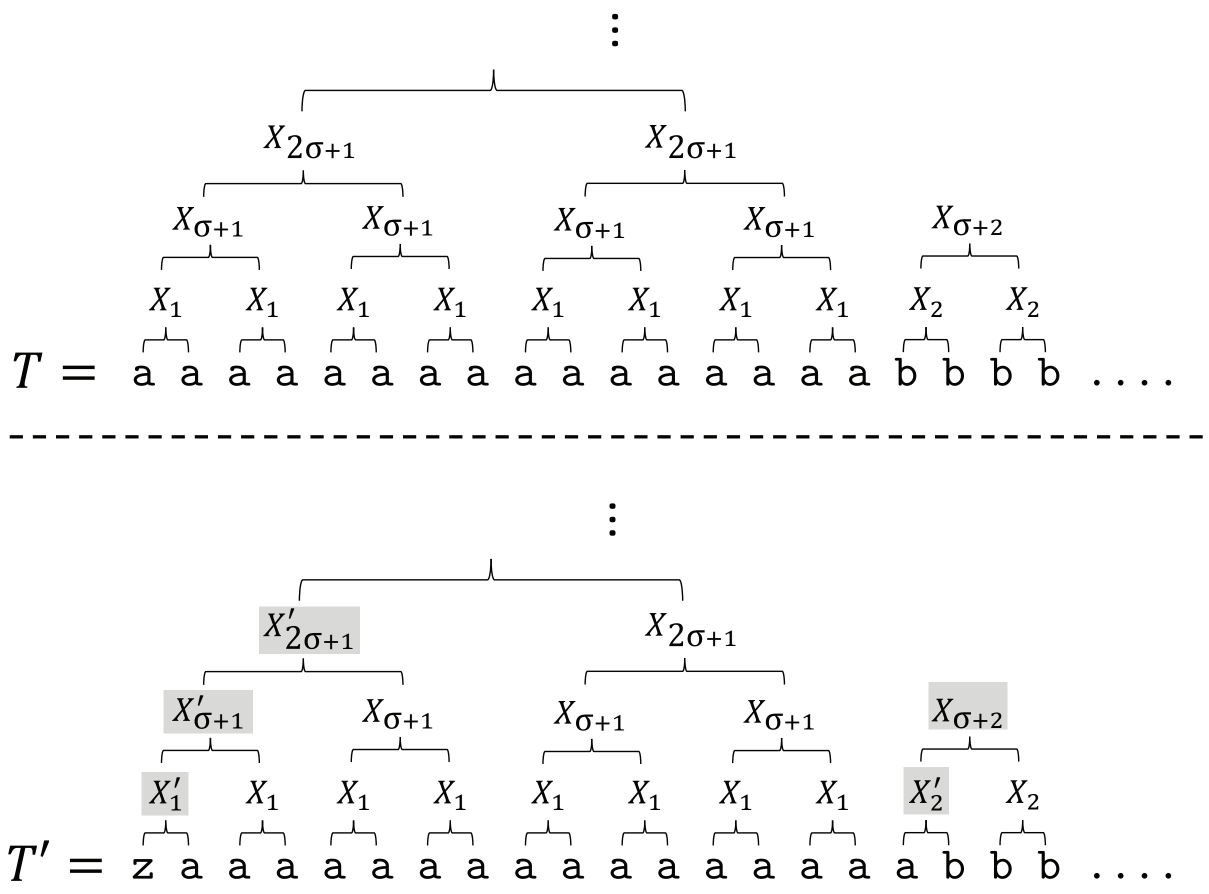


Figure 5.6: An example of insertion. There are nodes $X_1, X_{\sigma+1}, X_{2\sigma+1}, X_{3\sigma+1}$ in the leftmost path of in the derivation tree of $T = \mathtt{a}^{2^4}\mathtt{b}^{2^4}\mathtt{b}^{2^4}\cdots$. After a z is prepended to T $(= T')$, new internal nodes $X'_1, X'_{\sigma+1}, X'_{2\sigma+1}, X'_{3\sigma+1}$ that correspond to $\mathtt{za}, \mathtt{za}^3, \mathtt{za}^7, \mathtt{za}^{15}$ occur in the derivation tree for T'. This propagates to the other $\sigma - 1$ bigrams of form $\sigma_i\sigma_j$, which are ab, bc, and so forth.

5.2.2 Upper bounds for the sensitivity of g_{bsc}

Theorem 5.4. *The following upper bounds on the sensitivity of g_{bsc} hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \leq 2$. $\mathsf{AS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \leq 2\lceil \log_2 n \rceil \leq 2g_{\mathrm{bsc}}$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \leq \sigma + 1$. $\mathsf{AS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \leq \sigma g_{\mathrm{bsc}}$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \leq \sigma + 1$. $\mathsf{AS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \leq \sigma g_{\mathrm{bsc}}$.

Proof. **substitutions:** Let i be the position where we substitute the character $T[i]$. We consider the path P from the root of $\mathcal{T}$ to the ith leaf of $\mathcal{T}$ that corresponds to $T[i]$. We only need to change the labels of the nodes in the path P, since any other nodes do not contain the ith leaf. Since $\mathcal{T}$ is a balanced binary tree, the height h of $\mathcal{T}$ is $\lceil \log_2 n \rceil$ and hence $|P| \leq h = \lceil \log_2 n \rceil$. Since $h \leq g_{\mathrm{bsc}}$, we get $\mathsf{MS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \leq 2$. Since each non-terminal is in the Chomsky normal form and since $\lceil \log_2 n \rceil \leq g_{\mathrm{bsc}}$, $\mathsf{AS}_{\mathrm{sub}}(g_{\mathrm{bsc}}, n) \leq 2\lceil \log_2 n \rceil \leq 2g_{\mathrm{bsc}}$.

insertions: Let i be the position where we insert a new character a to T, and let $\mathcal{T}$ and $\mathcal{T}'$ be the derivation trees for the strings T and T' before and after the insertion, respectively. For any node v in the derivation tree $\mathcal{T}$, let $\mathcal{T}(v)$ denote the subtree rooted at v. Let $\ell(v)$ and $r(v)$ denote the text positions that respectively correspond to the leftmost and rightmost leaves in $\mathcal{T}(v)$. We use the same analysis for the left children of the nodes in the path P from the root to the new ith leaf which corresponds to the inserted character a. Let v' denote a node in $\mathcal{T}'$. From now on let us focus on the subtrees $\mathcal{T}'(v')$ of $\mathcal{T}'$ such that $\ell(v') > i$ and v' is not in the rightmost path from the root of $\mathcal{T}'$. Let $str(v')$ denote the string that is derived from the non-terminal for v', and let v be the node in $\mathcal{T}$ which corresponds to v'. Observe that $str(v') = T'[\ell(v')..r(v')] = T[\ell(v) - 1..r(v) - 1]$, namely, $str(v')$ has been shifted by one position in the string due to the new character a inserted at position i. Since $T[\ell(v)..r(v)]$ is represented by the node v in $\mathcal{T}$, there exist at most g_{bsc} distinct substrings of T that can be the "seed" of the strings represented by the nodes v' of $\mathcal{T}'$ with $\ell(v') > i$. Since the number of left-contexts of each $T[\ell(v)..r(v)]$ is at most σ, there can be at most σ distinct shifts from the seed $T[\ell(v)..r(v)]$. Since the rightmost paths from the roots of $\mathcal{T}$ and $\mathcal{T}'$ are all distinct except the root, and since inserting the character can increase the length of the rightmost path by at most 1, overall, we have that

$$g_{\mathrm{bsc}}(T') \leq \sigma g_{\mathrm{bsc}}(T) + \lceil \log_2 n \rceil + 1 \leq \sigma g_{\mathrm{bsc}}(T) + h + 1, \tag{5.1}$$

where h is the height of $\mathcal{T}$. To analyze an upper bound, we can exclude the case of unary alphabets with $T = a^n$ and $T' = a^{n+1}$, since the sensitivity for Bisection is the minimum possible with unary strings for insertions. For the case of multi-character alphabets $g_{\mathrm{bsc}} \geq$

$h + 1$ holds, and hence $g_{\mathrm{bsc}}(T') \leq (\sigma + 1)g_{\mathrm{bsc}}(T)$ follows from formula (5.1). Hence we get $\mathsf{MS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \leq \sigma + 1$ and $\mathsf{AS}_{\mathrm{ins}}(g_{\mathrm{bsc}}, n) \leq \sigma g_{\mathrm{bsc}}$.

deletions: By similar arguments to the case of insertions, we get $\mathsf{MS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \leq \sigma + 1$ and $\mathsf{AS}_{\mathrm{del}}(g_{\mathrm{bsc}}, n) \leq \sigma g_{\mathrm{bsc}}$.

$\square$

5.3 LZ-End factorizations

In this section, we consider the worst-case sensitivity of the *LZ-End factorizations* [69]. This is an LZ77-like compressor such that each factor f_i has a previous occurrence which corresponds to the ending position of a previous factor. This property allows for fast substring extraction in practice [69].

A factorization $T = f_1 \cdots f_{z_{\text{End}}}$ for a string T of length n is the LZ-End factorization $\mathsf{LZEnd}(T)$ of T such that, for each $1 \leq i < z_{\text{End}}$, $f_i[1..|f_i|-1]$ is the longest prefix of $f_i \cdots f_{z_{\text{End}}}$ which has a previous occurrence in $f_1 \cdots f_{i-1}$ as a suffix of some string in $\{\varepsilon, f_1, f_1 f_2, \ldots,$ $f_1 \cdots f_{i-1}\}$. The last factor $f_{z_{\text{End}}}$ is the suffix of T of length $n - |f_1 \cdots f_{z_{\text{End}}-1}|$. Again, if we use a common convention that the string T terminates with a unique character \$, then the last factor $f_{z_{\text{End}}}$ satisfies the same properties as $f_1, \ldots, f_{z-1}$, in the cases of LZ-End factorizations. Let $z_{\text{End}}(T)$ denote the number of factors in the LZ-End factorization of string T.

For example, for string $T = \mathtt{abaababababababab\$}$,

$$T = \mathtt{a|b|aa|ba|bab|ababab\$|},$$

where $|$ denotes the right-end of each factor in the factorization. Here we have $z_{\text{End}}(T) = 6$.

5.3.1 Lower bounds for the sensitivity of z_{End}

Theorem 5.5. *The following lower bounds on the sensitivity of z_{End} hold:*
substitutions: $\liminf_{n\to\infty} \mathsf{MS}_{\text{sub}}(z_{\text{End}}, n) \geq 2$. $\mathsf{AS}_{\text{sub}}(z_{\text{End}}, n) \geq z_{\text{End}} - \Theta(\sqrt{z_{\text{End}}})$ *and* $\mathsf{AS}_{\text{sub}}(z_{\text{End}}, n) \in \Omega(\sqrt{n})$.
insertions: $\liminf_{n\to\infty} \mathsf{MS}_{\text{ins}}(z_{\text{End}}, n) \geq 2$. $\mathsf{AS}_{\text{ins}}(z_{\text{End}}, n) \geq z_{\text{End}} - \Theta(\sqrt{z_{\text{End}}})$ *and* $\mathsf{AS}_{\text{ins}}(z_{\text{End}}, n) \in \Omega(\sqrt{n})$.
deletions: $\liminf_{n\to\infty} \mathsf{MS}_{\text{del}}(z_{\text{End}}, n) \geq 2$. $\mathsf{AS}_{\text{del}}(z_{\text{End}}, n) \geq z_{\text{End}} - \Theta(\sqrt{z_{\text{End}}})$ *and* $\mathsf{AS}_{\text{del}}(z_{\text{End}}, n) \in \Omega(\sqrt{n})$.

Proof. Let σ_j denote the ith character in the alphabet Σ for $1 \leq j \leq |\Sigma|$. For a positive integer p, consider the string $Q = \sigma_1 \cdot \sigma_1\sigma_2 \cdots \sigma_1 \cdots \sigma_p$ of length $q = |Q| = p(p+1)/2 \in \Theta(p^2)$. Consider the string

$$T = Q \cdot \sigma_1\sigma_{p+1} \cdot Q[q]\sigma_1\sigma_{p+1}\sigma_{p+2} \cdot Q[q-1..q]\sigma_1\sigma_{p+1}\sigma_{p+3} \cdots Q\sigma_1\sigma_{p+1}\sigma_{p+q+1}$$

with $|T| \in \Theta(q^2)$. As for the interval $[1, q]$ in T, $\mathsf{LZEnd}(Q) = f_1, \ldots, f_p$ such that $f_k = \sigma_1 \cdots \sigma_k$ for every $1 \leq k \leq p$. Since f_p has no occurrences in $f_1 \cdots f_{p-1}$, the decomposition is not changed by appending any character to Q. Hence, the next factor f_{p+1} starts at position $q+1$. Then $f_{p+1} = \sigma_1\sigma_{p+1}$ holds, since σ_{p+1} is a fresh character and $\sigma_1 = f_1$. As for the remaining interval, we show $f_{p+1+j} = Q[q-j+1..q]\sigma_1\sigma_{p+1}\sigma_{p+j+1}$ holds for each $1 \le j \le q$. At first, for $j = 1$, $f_{p+2} = Q[q]\sigma_1\sigma_{p+1}\sigma_{p+2}$ holds since f_{p+2} starts with $Q[q]$, $Q[q]\sigma_1\sigma_{p+1}$ has an occurrence as a suffix of $T[1..q+2]$, and σ_{p+2} is a fresh character in the prefix. Next, we assume that $f_{p+1+j} = Q[q-j+1..q]\sigma_1\sigma_{p+1}\sigma_{p+j+1}$ holds with $1 \le j \le k-1$ for some integer k. Then we consider whether $f_{p+1+k} = Q[q-k+1..q]\sigma_1\sigma_{p+1}\sigma_{p+k+1}$ holds or not. By the assumption, f_{p+1+k} starts with $Q[q-k+1]$. Also, $Q[q-k+1..q]\sigma_1\sigma_{p+1}$ has an occurrence as a suffix of $T[1..q+2]$, and σ_{p+k+1} is a fresh character in the prefix. Therefore, the assumption is also valid for k. By the above argument, $f_{p+1+j} = Q[q-j+1..q]\sigma_1\sigma_{p+1}\sigma_{p+j+1}$ holds for each $1 \le j \le q$ by induction. Therefore,

$$\mathsf{LZEnd}(T) = \sigma_1|\sigma_1\sigma_2|\cdots|\sigma_1\cdots\sigma_p|\sigma_1\sigma_{p+1}|Q[q]\sigma_1\sigma_{p+1}\sigma_{p+2}|\cdots|Q\sigma_1\sigma_{p+1}\sigma_{p+q+1}|$$

with $z_{\text{End}}(T) = p+1+q \in \Theta(q) \in \Theta(\sqrt{n})$.

As for substitutions, consider the string

$$T' \;=\; Q \cdot \#\sigma_{p+1} \cdot Q[q]\sigma_1\sigma_{p+1}\sigma_{p+2} \cdot Q[q-1..q]\sigma_1\sigma_{p+1}\sigma_{p+3}\cdots Q\sigma_1\sigma_{p+1}\sigma_{p+q+1}$$

which can be obtained from T by substituting $T[q+1] = \sigma_1$ with a character $\#$ which does not occur in T. Let us analyze the structure of the $\mathsf{LZEnd}(T')$. As mentioned above, $f_1, \ldots, f_p$ are not changed after the substitution. The $(p+1)$th factor in $\mathsf{LZEnd}(T)$, namely, $\#\sigma_{p+1}$ is factorized as $\#|\sigma_{p+1}|$ in $\mathsf{LZEnd}(T')$ since both characters have no occurrence in $T'[1..q] = Q$. Then the next factor starts with $Q[q]$. Each of $Q[q]$ and σ_1 have some occurrence as a suffix of a previous factor. On the other hand, each of $Q[q]\sigma_1$ and $\sigma_{p+1}\sigma_{p+2}$ have no occurrences previously. Therefore, $Q[q]\sigma_1\sigma_{p+1}\sigma_{p+2}$ is factorized as $Q[q]\sigma_1|\sigma_{p+1}\sigma_{p+2}|$ in $\mathsf{LZEnd}(T')$. Similarly, by induction, each $(p+1+k)$th factor in $\mathsf{LZEnd}(T)$ for every $2 \le k \le q$, namely, $Q[q-k+1..q]\sigma_1\sigma_{p+1}\sigma_{p+k+1}$ is also factorized as $Q[q-k+1..q]\sigma_1|\sigma_{p+1}\sigma_{p+k+1}|$. Thus, the LZ-End factorization of T' is

$$\mathsf{LZEnd}(T') = \sigma_1|\sigma_1\sigma_2|\cdots|\sigma_1\cdots\sigma_p|\#|\sigma_{p+1}|Q[q]\sigma_1|\sigma_{p+1}\sigma_{p+2}|\cdots|Q\sigma_1|\sigma_{p+1}\sigma_{p+q+1}|$$

with $z_{\text{End}}(T') = p+2+2q$. Recall $p = \Theta(\sqrt{q})$. Hence we get $\liminf_{n\to\infty} \mathsf{MS}_{\text{sub}}(z_{\text{End}}, n) \ge \liminf_{q\to\infty}(p+2q+2)/(p+q+1) \ge 2$, $\mathsf{AS}_{\text{sub}}(z_{\text{End}}, n) \ge (p+2q+2)-(p+q+1) = z_{\text{End}} - \Theta(\sqrt{z_{\text{End}}})$, and $\mathsf{AS}_{\text{sub}}(z_{\text{End}}, n) \in \Omega(\sqrt{n})$.

Also, as for deletions (resp. insertions), we get Theorem 5.5 by considering the case where the character $T[q+1]$ is deleted (resp. $\#$ is inserted between positions q and $q+1$). $\square$

5.3.2 Upper bounds for the sensitivity of z_{End}

To show a non-trivial upper bound for the sensitivity of z_{End}, we use the following known results:

Theorem 5.6 ([69])**.** *For any string T, $z_{\mathrm{SSsr}}(T) \le z_{\mathrm{End}}(T)$.*

Theorem 5.7 (Theorem 3.2 of [62])**.** *For any string T of length n, $z_{\mathrm{End}}(T) \in O(\delta(T)\log^2(n/\delta(T)))$.*

Theorem 5.8 (Lemma 3.7 of [61])**.** *For any string T, $\gamma(T) \le z_{\mathrm{SSsr}}(T)$.*

Theorem 5.9 (Lemma 2 of [66])**.** *For any string T, $\gamma(T) \ge \delta(T)$.*

Theorem 5.10 ([1])**.** *The following upper bounds on the sensitivity of δ hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(\delta, n) \le 2$. $\mathsf{AS}_{\mathrm{sub}}(\delta, n) \le 1$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(\delta, n) \le 2$. $\mathsf{AS}_{\mathrm{ins}}(\delta, n) \le 1$.
deletions: $\limsup_{n\to\infty} \mathsf{MS}_{\mathrm{del}}(\delta, n) \le 1.5$. $\limsup_{n\to\infty} \mathsf{AS}_{\mathrm{del}}(\delta, n) \le 1$.

From Theorems 5.8, 5.9, 5.7, and 5.6, we obtain the following result:

Corollary 5.1. *The following upper bounds on the sensitivity of z_{End} hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(z_{\mathrm{End}}, n) \in O(\log^2(n/\delta))$. $\mathsf{AS}_{\mathrm{sub}}(z_{\mathrm{End}}, n) \in O(z_{\mathrm{End}}\log^2(n/\delta))$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(z_{\mathrm{End}}, n) \in O(\log^2(n/\delta))$. $\mathsf{AS}_{\mathrm{ins}}(z_{\mathrm{End}}, n) \in O(z_{\mathrm{End}}\log^2(n/\delta))$.
deletions: $\mathsf{MS}_{\mathrm{del}}(z_{\mathrm{End}}, n) \in O(\log^2(n/\delta))$. $\mathsf{AS}_{\mathrm{del}}(z_{\mathrm{End}}, n) \in O(z_{\mathrm{End}}\log^2(n/\delta))$.

Proof. For any string T, $\delta(T) \le z_{\mathrm{End}}(T)$ holds from Theorems 5.9, 5.8, and 5.6. Let T' be any string with $\mathsf{ed}(T, T') = 1$. It follows from Theorem 5.10 that $\delta(T') \le 2\delta(T)$. Now let c be the constant value such that $\delta(T') = c\delta(T)$ holds. Then, $\log^2(n/\delta(T')) = \log^2 n + \log^2 c\delta(T) - 2\log n \log c\delta(T) = \log^2 n + \log^2 \delta(T) - 2\log n \log\delta(T) + \log^2 c + 2\log\delta(T)\log c - 2\log n \log c \in O(\log^2(n/\delta(T)))$. Following Lemma 2.1, we now obtain
$z_{\mathrm{End}}(T') \in O(\delta(T')\log^2(n/\delta(T'))) \in O(\delta(T)\log^2(n/\delta(T))) \in O(z_{\mathrm{End}}(T)\log^2(n/\delta(T)))$, which leads to the claimed upper bounds for the sensitivity for z_{End}. □

5.4 Smallest grammars and approximation grammars

In this section, we consider the sensitivity of the smallest grammar size g^* and several grammars whose sizes satisfy some approximation ratios to g^*.

5.4.1 Smallest grammar

In this section (and also in the following sections), we consider *grammar-based* compressors for input string T.

It is known that the problem of computing the size $g^*(T)$ of the smallest grammar only generating T is NP-hard [101, 21]. It is also known that $z_{\mathrm{SS}}(T)$ is a lower bound of the size of *any* grammar generating T, namely, $z_{\mathrm{SS}}(T) \leq g^*(T)$ holds for any string T [98, 21].

We have the following upper bounds for the sensitivity of $g^*(T)$:

Theorem 5.11. *The following upper bounds on the sensitivity of g^* hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g^*, n) \leq 2$. $\mathsf{AS}_{\mathrm{sub}}(g^*, n) \leq g^*$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g^*, n) \leq 2$. $\mathsf{AS}_{\mathrm{ins}}(g^*, n) \leq g^*$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g^*, n) \leq 2$. $\mathsf{AS}_{\mathrm{del}}(g^*, n) \leq g^*$.

Proof. Let T be any string of length n, and let $\mathcal{G}^*(T)$ be a grammar of size $g^*(T)$ that only generates T.

We describe the case of substitutions. Let T' be the string that can be obtained by substituting a character c for the ith character $T[i]$ of T, where $c \neq T[i]$. Let X be a non-terminal of $\mathcal{G}^*(T)$ in the path P from the root to the leaf for the ith character in the derivation tree of $\mathcal{G}^*(T)$. Let $X \to Y_1 \cdots Y_k$ be the production from X, and let Y_j $(1 \leq j \leq k)$ be the non-terminal that is the child of X in the path P. Then, we introduce a new non-terminal X' and a new production $X' \to Y_1 \cdots Y_{j-1} Y'_j Y_{j+1} \cdots Y_k$, where Y'_j will be the new non-terminal at the next depth in the path P. By applying this operation in a top-down manner on P, we can obtain a grammar $\mathcal{G}(T')$ of size $g(T') \leq 2g^*(T)$ that generates T'. Since $g^*(T') \leq g(T')$, we have the claimed bounds. The cases with insertions and deletions are analogous. □

5.4.2 Practical grammars

Since computing a smallest grammar of size $g^*(T)$ is NP-hard, a number of practical grammar-based compressors have been proposed, including RePair [72], Longest-Match [64], Greedy [4], Sequential [108][1], and LZ78 [111][2]. Charikar et al. [21] analyzed the approximation ratios of these grammar compressors to the smallest grammar. Let g_{rpair}, g_{long}, g_{grdy}, g_{seq}, z_{78} denote the sizes of the aforementioned compressors, respectively. It is known that for any $g \in \{g_{\mathrm{rpair}}, g_{\mathrm{long}}, g_{\mathrm{grdy}}, z_{78}\}$ $g(T) \in O(g^*(T)(n/\log n)^{\frac{2}{3}})$ holds, and $g_{\mathrm{seq}} \in O(g^*(T)(n/\log n)^{\frac{3}{4}})$ holds [21]. By combining these results with Lemma 2.1 and Theorem 5.11, we obtain the following bounds:

Corollary 5.2. *The following upper bounds for the sensitivity of* $g \in \{g_{\mathrm{rpair}}, g_{\mathrm{long}}, g_{\mathrm{grdy}}, z_{78}\}$ *hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g,n) \in O((n/\log n)^{\frac{2}{3}})$. $\mathsf{AS}_{\mathrm{sub}}(g,n) \in O(g^* \cdot (n/\log n)^{\frac{2}{3}})$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g,n) \in O((n/\log n)^{\frac{2}{3}})$. $\mathsf{AS}_{\mathrm{ins}}(g,n) \in O(g^* \cdot (n/\log n)^{\frac{2}{3}})$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g,n) \in O((n/\log n)^{\frac{2}{3}})$. $\mathsf{AS}_{\mathrm{del}}(g,n) \in O(g^* \cdot (n/\log n)^{\frac{2}{3}})$.

Corollary 5.3. *The following upper bounds for the sensitivity of* g_{seq} *hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g_{\mathrm{seq}},n) \in O((n/\log n)^{\frac{3}{4}})$. $\mathsf{AS}_{\mathrm{sub}}(g_{\mathrm{seq}},n) \in O(g^* \cdot (n/\log n)^{\frac{3}{4}})$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g_{\mathrm{seq}},n) \in O((n/\log n)^{\frac{3}{4}})$. $\mathsf{AS}_{\mathrm{ins}}(g_{\mathrm{seq}},n) \in O(g^* \cdot (n/\log n)^{\frac{3}{4}})$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g_{\mathrm{seq}},n) \in O((n/\log n)^{\frac{3}{4}})$. $\mathsf{AS}_{\mathrm{del}}(g_{\mathrm{seq}},n) \in O(g^* \cdot (n/\log n)^{\frac{3}{4}})$.

5.4.3 Approximation grammars

There also exist (better) approximation algorithms in terms of the smallest grammar size g^*.

It is known that α-balanced grammar compressor [21], the AVL-grammar compressor [98], and the really-simple grammar compressor [59] all achieve $O(\log(n/g^*))$-approximation ratios to g^*. Let g_α, g_{avl}, and g_{simple} denote the sizes of these compressors, respectively. Namely, for every $g \in \{g_\alpha, g_{\mathrm{avl}}, g_{\mathrm{simple}}\}$, $g \in O(g^* \log(n/g^*))$ holds. Since $\log(n/g^*)$ satisfies the conditions for the function $f(n, g^*)$ in Lemma 2.1, and since g^* satisfies the conditions Lemma 2.1 by Theorem 5.11, we obtain the following:

Corollary 5.4. *The following upper bounds for the sensitivity of* $g \in \{g_\alpha, g_{\mathrm{avl}}, g_{\mathrm{simple}}\}$ *hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g,n) \in O(\log(n/g^*))$. $\mathsf{AS}_{\mathrm{sub}}(g,n) \in O(g^* \log(n/g^*))$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g,n) \in O(\log(n/g^*))$. $\mathsf{AS}_{\mathrm{ins}}(g,n) \in O(g^* \log(n/g^*))$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g,n) \in O(\log(n/g^*))$. $\mathsf{AS}_{\mathrm{del}}(g,n) \in O(g^* \log(n/g^*))$.

We remark that $z_{\mathrm{SS}}(T) \leq g^*(T)$ always holds [98], and that α-balanced grammars and AVL-grammars are known to achieve approximation ratios to $z_{\mathrm{SS}}(T)$. Namely, it is known that $g_\alpha(T) \in O(z_{\mathrm{SS}}(T)\log n)$ and $g_{\mathrm{avl}}(T) \in O(z_{\mathrm{SS}} \log n)$ hold [21, 98].

[1]Sequential is an improved version of Sequitur [88].
[2]The LZ78 factorization can also be seen as a grammar.

Let f_i be a factor of the LZSS factorization of T such that f_i has previous occurrence(s) in $f_1 \cdots f_{i-1}$. When we compress the string T with LZSS, f_i is encoded as a pair (b, e) such that $T[b..e] = f_i$ and $1 \leq b \leq e < |f_1 \ldots f_{i-1}|$. The substring $T[b..e]$ is called the *source* for f_i.

Corollary 5.5. *The following upper bounds for the sensitivity of $g \in \{g_\alpha, g_{\mathrm{avl}}\}$ hold:*
substitutions: $\mathsf{MS}_{\mathrm{sub}}(g, n) \in O(\log n)$. $\mathsf{AS}_{\mathrm{sub}}(g, n) \in O(z_{\mathrm{SS}} \log n)$.
insertions: $\mathsf{MS}_{\mathrm{ins}}(g, n) \in O(\log n)$. $\mathsf{AS}_{\mathrm{ins}}(g, n) \in O(z_{\mathrm{SS}} \log n)$.
deletions: $\mathsf{MS}_{\mathrm{del}}(g, n) \in O(\log n)$. $\mathsf{AS}_{\mathrm{del}}(g, n) \in O(z_{\mathrm{SS}} \log n)$.

Proof. $z_{\mathrm{SS}}(T') \leq 3z_{\mathrm{SS}}(T)$ holds for any string T' with $\mathsf{ed}(T, T') = 1$ [1]. The edit operation that transforms T into T' may change the positions of the sources in the LZSS factorizations (e.g., if the LZSS compression is implemented so that the source of each factor is always its leftmost occurrence, then the three divided fragments $f'_{i_1}, f'_{i_2}, f'_{i_3}$ can have more left previous occurrences than the original factor f_i). However, an important observation is that the $O(\log n)$-approximation proofs for g given in [21, 98] hold for *any* choices of the sources for the factors. Thus, we obtain $g(T') \in O(z_{\mathrm{SS}}(T') \log n) \subseteq O(z_{\mathrm{SS}}(T) \log n)$.

By combining it with $z_{\mathrm{SS}}(T) \leq g(T)$ and by Lemma 2.1, we get $g(T')/g(T) \leq g(T')/z_{\mathrm{SS}}(T) \in O(\log n)$. Similarly we obtain $g(T') - g(T) \leq g(T') - z_{\mathrm{SS}}(T) \in O(z_{\mathrm{SS}}(T) \log n)$. $\square$

We remark that the $O(\log n)$-bounds for the multiplicative sensitivity of Theorem 5.4 can also be obtained using the smallest grammar size g^*. However, the $O(z_{\mathrm{SS}} \log n)$-bounds for the additive sensitivity of Theorem 5.4 can be better than the $O(g^* \log(n/g^*))$-bounds obtained via g^*, since $z_{\mathrm{SS}} \leq g^*$ always holds [98].

Chapter 6

Conclusions and future work

In Chapter 4, we discovered a local consistency property in parsing of the grammar produced by the grammar compression by induced sorting (GCIS) [94], and apply it to our new grammar index. On a theoretical level, we show we can compute the index from T of length n in $O(n)$ time, and the index locate all occ occurrences of a given pattern of length m in $O(m \lg |\mathcal{S}| + \mathrm{occ}_C \lg m \lg |\mathcal{S}| + \mathrm{occ})$ time, where $\mathcal{S}$ is the set of characters and non-terminals of the GCIS grammar and occ_C the number of occurrences in the right side of the production rules of the GCIS grammar of a selected core of the pattern. In particular for experiments of ENGLISH.001.2, our implementations are faster than FM-index[40] and r-index[47] when the pattern length reaches 10000 characters or more. In addition, GCIS can form cores that cover a higher percentage of patterns than cores selected by ESP, so GCIS-index can compute locate faster than ESP-index in experiments of FIB41.

In Chapter 5, we show the upper bounds and lower bounds for the sensitivity of some compressors. In the seminal paper by Varma and Yoshida [106] which first introduced the notion of sensitivity for (general) algorithms and studied the sensitivity of graph algorithms, the authors wrote:

> "*Although we focus on graphs here, we note that our definition can also be extended to the study of combinatorial objects other than graphs such as strings and constraint satisfaction problems.*"

Our study was inspired by the afore-quoted suggestion, and our sensitivity for string compressors and repetitiveness measures enables one to evaluate the robustness and stability of compressors and repetitiveness measures.

The major technical contributions of this thesis are the *tight and constant upper and lower bounds* for the multiplicative sensitivity of the GCIS and the smallest bidirectional scheme b.

We also reported non-trivial upper and/or lower bounds for other string compressors, including RLBWT, LZ-End, LZ78, AVL-grammar, α-balanced grammar, RePair, LongestMatch, Greedy, Bisection, and CDAWG. Some of the upper bounds reported here follow from previous important work [61, 66, 60, 69, 62, 21, 98, 59].

Apparent future work is to complete Tables 1.1 and 1.2 by filling the missing pieces and closing the gaps between the upper and lower bounds which are not tight there. In addition, the research for the relation of sensitivity between GCIS and our new Index is also counted a future topic. If It is proved, GCIS Index has a robust compression ratio against single character edits. Then, devising a new algorithm for GCIS that can handle small substitutions will become more important future work.

While we dealt with a number of string compressors and repetitiveness measures, it has to be noted that our list is far from being comprehensive: It is intriguing to analyze the sensitivity of other important and useful compressors and repetitiveness measures including the size ν of the smallest NU-systems [86], the sizes of the other locally-consistent compressed indices such as ESP-index [77] and SE-index [89].

Our notion of the sensitivity for string compressors/repetitiveness measures can naturally be extended to labeled tree compressors/repetitiveness measures. It would be interesting to analyze the sensitivity for the smallest tree attractor [96], the run-length XBWT [96], the tree LZ77 factorization [52], tree grammars [73, 49], and top-tree compression of trees [14].

In Chapter 3, we discussed Minimal absent words (MAWs), which is combinatorial string objects that can be used in applications such as data compression (anti-dictionaries) and bioinformatics.

In the first section of Chapter 3, we also revisited the problem of computing the minimal absent words (MAWs) for the sliding window model, which was first considered by Crochemore et al. [30]. We investigated combinatorial properties of MAWs for a sliding window of fixed length d over a string of length n. Our contributions are *matching upper and lower bounds* for the number of changes in the set of MAWs for a sliding window when the window is shifted to the right by one character. For the general case where the window S and the extended window $S\alpha$ contain three or more distinct characters (i.e. $\sigma' \geq 3$), the number of changes in the set of MAWs for S and $S\alpha$ is at most $d + \sigma' + 1$ and this bound is tight. For the case of binary alphabets (i.e. $\sigma' = 2$), it is upper bounded by $\max\{3, d\}$ and this bound is also tight.

We eventually gave an asymptotically tight bound $O(\min\{d, \sigma\}n)$ for the number $\mathcal{S}(T, d)$ of total changes in the set of MAWs for every sliding window of length d over any string T of length n, where σ is the alphabet size for the whole input string T.

An interesting open question is whether there exist other compressed representations of MAWs, based on e.g. grammar-based compression [64], Lempel-Ziv 77 [110], and run-length Burrows-Wheeler transform [74]. About the theme of the sliding window, the following open questions are intriguing:

- We showed that a matching lower bound $\mathcal{S}(T, d) \in \Omega(\min\{d, \sigma\}n)$ when $n - d \in \Omega(n)$. Is there a similar lower bound when $n - d \in o(n)$?
- Crochemore et al. [30] gave an online algorithm that maintains the set of MAWs for a sliding window of length d in $O(\sigma n)$ time. Can one improve the running time to optimal $O(\min\{d, \sigma\}n)$?

In the second section of Chapter 3, we considered MAWs for a string T that is described by its run-length encoding (RLE) $\mathsf{rle}(T)$ of size m. We first analyzed the number of MAWs for a string T in terms of its RLE size m, by dividing the set $\mathsf{MAW}(T)$ of all MAWs for T into five disjoint types. Albeit the number of MAWs of some types is superlinear in m, we devised a compact $O(m)$-space representation for $\mathsf{MAW}(T)$ that can output all MAWs in output-sensitive $O(|\mathsf{MAW}(T)|)$ time. Additionally we show the algorithms to construct the representation for each type MAWs in totally $O(m)\log m$ time. A simple future work of this part is to analyze the amount of MAW in the other compressors. In this thesis, we proposed a representation approach that takes just $O(m)$ space even when the number of some types (3 and 4) much greater than $O(m)$. This technique might be a clue for finding compact MAW representations for other compression methods.

Bibliography

[1] T. Akagi, M. Funakoshi, and S. Inenaga. Sensitivity of string compressors and repetitiveness measures. *Information and Computation*, page 104999, 2022.

[2] T. Akagi, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Grammar index by induced suffix sorting. In *SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 2021.

[3] Y. Almirantis, P. Charalampopoulos, J. Gao, C. S. Iliopoulos, M. Mohamed, S. P. Pissis, and D. Polychronopoulos. On avoided words, absent words, and their application to biological sequence analysis. *Algorithms for Molecular Biology*, 12(1):5, 2017.

[4] A. Apostolico and S. Lonardi. Off-line compression by greedy textual substitution. *Proceedings of the IEEE*, 88(11):1733–1744, 2000.

[5] H. Bannai, T. Gagie, and T. I. Refining the r-index. *Theor. Comput. Sci.*, 812:96–108, 2020.

[6] H. Bannai, K. Goto, M. Ishihata, S. Kanda, D. Köppl, and T. Nishimoto. Computing np-hard repetitiveness measures via max-sat. *arXiv preprint arXiv:2207.02571*, 2022.

[7] 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.

[8] C. Barton, A. Heliou, L. Mouchard, and S. P. Pissis. Linear-time computation of minimal absent words using suffix array. *BMC Bioinformatics*, 15(1):388, 2014.

[9] C. Barton, A. Heliou, L. Mouchard, and S. P. Pissis. Parallelising the computation of minimal absent words. In *PPAM 2015*, pages 243–253, 2016.

[10] D. Belazzougui, M. Cáceres, T. Gagie, P. Gawrychowski, J. Kärkkäinen, G. Navarro, A. O. Pereira, S. J. Puglisi, and Y. Tabei. Block trees. *J. Comput. Syst. Sci.*, 117:1–22, 2021.

[11] D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen. Versatile succinct representations of the bidirectional Burrows-Wheeler transform. In *ESA 2013*, pages 133–144, 2013.

[12] D. Belazzougui, T. Gagie, P. Gawrychowski, J. Kärkkäinen, A. O. Pereira, S. J. Puglisi, and Y. Tabei. Queries on lz-bounded encodings. In *2015 Data Compression Conference, DCC 2015, Snowbird, UT, USA, April 7-9, 2015*, pages 83–92, 2015.

[13] P. Bille, M. B. Ettienne, I. L. Gørtz, and H. W. Vildhøj. Time-space trade-offs for Lempel-Ziv compressed indexing. *Theor. Comput. Sci.*, 713:66–77, 2018.

[14] P. Bille, P. Gawrychowski, I. L. Gørtz, G. M. Landau, and O. Weimann. Top tree compression of tries. In *ISAAC 2019*, volume 149 of *LIPIcs*, pages 4:1–4:18, 2019.

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

[16] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. STOC*, pages 279–288, 1998.

[17] M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

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

[19] P. Charalampopoulos, M. Crochemore, G. Fici, R. Mercaş, and S. P. Pissis. Alignment-free sequence comparison using absent words. *Information and Computation*, 262:57–68, 2018.

[20] P. Charalampopoulos, M. Crochemore, and S. P. Pissis. On extended special factors of a word. In *SPIRE 2018*, pages 131–138. Springer, 2018.

[21] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005.

[22] A. R. Christiansen, M. B. Ettienne, T. Kociumaka, G. Navarro, and N. Prezza. Optimal-time dictionary-compressed indexes. *ACM Trans. Algorithms*, 17(1):8:1–8:39, 2021.

[23] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Universal indexes for highly repetitive document collections. *Inf. Syst.*, 61:1–23, 2016.

[24] F. Claude and G. Navarro. Self-indexed grammar-based compression. *Fundam. Inform.*, 111(3):313–337, 2011.

[25] F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. SPIRE*, volume 7608 of *LNCS*, pages 180–192, 2012.

[26] F. Claude, G. Navarro, and A. Pacheco. Grammar-compressed indexes with logarithmic search time. *J. Comput. Syst. Sci.*, 118:53–74, 2021.

[27] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007.

[28] T. Crawford, G. Badkobeh, and D. Lewis. Searching page-images of early music scanned with OMR: A scalable solution using minimal absent words. In *ISMIR 2018*, pages 233–239, 2018.

[29] M. Crochemore. Linear searching for a square in a word. *Bulletin of the European Association of Theoretical Computer Science*, 24:66–72, 1984.

[30] M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis, and Y. Ramusat. Absent words in a sliding window with applications. *Information and Computation*, 270:104461, 2020.

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

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

[33] M. Crochemore and G. Navarro. Improved antidictionary based compression. In *12th International Conference of the Chilean Computer Science Society, 2002. Proceedings.*, pages 7–13. IEEE, 2002.

[34] D. Díaz-Domínguez and G. Navarro. A grammar compressor for collections of reads with applications to the construction of the BWT. In *Proc. DCC*, pages 83–92, 2021.

[35] D. Díaz-Domínguez, G. Navarro, and A. Pacheco. An LMS-based grammar self-index with local consistency properties. In T. Lecroq and H. Touzet, editors, *SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 2021.

[36] C. F. Du, H. Mousavi, L. Schaeffer, and J. O. Shallit. Decision algorithms for fibonacci-automatic words, with applications to pattern avoidance. *CoRR*, abs/1406.0670, 2014.

[37] P. Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2):246–260, 1974.

[38] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory*, 21(2):194–203, 1975.

[39] H. Ferrada and G. Navarro. Lempel-Ziv compressed structures for document retrieval. *Inf. Comput.*, 265:1–25, 2019.

[40] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13:1.12:1 – 1.12:31, 2008.

[41] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

[42] G. Fici and P. Gawrychowski. Minimal absent words in rooted and unrooted trees. In *Proc. SPIRE*, volume 11811 of *LNCS*, pages 152–161, 2019.

[43] J. Fischer, T. I, and D. Köppl. Deterministic sparse suffix sorting in the restore model. *ACM Trans. Algorithms*, 16(4):50:1–50:53, 2020.

[44] Y. Fujishige, Y. Tsujimaru, S. Inenaga, H. Bannai, and M. Takeda. Computing DAWGs and minimal absent words in linear time for integer alphabets. In *MFCS 2016*, volume 58, pages 38:1–38:14, 2016.

[45] 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.

[46] T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto, and Y. Takabatake. Rpair: Rescaling RePair with Rsync. *CoRR*, abs/1906.00809, 2019.

[47] T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in BWT-runs bounded space. In *Proc. SODA*, pages 1459–1477, 2018.

[48] T. Gagie, G. Navarro, and N. Prezza. Fully functional suffix trees and optimal text searching in BWT-runs bounded space. *J. ACM*, 67(1):2:1–2:54, 2020.

[49] M. Ganardi, D. Hucke, M. Lohrey, and E. Noeth. Tree compression using string grammars. *Algorithmica*, 80(3):885–917, 2018.

[50] L. Gasieniec and W. Rytter. Almost optimal fully LZW-compressed pattern matching. In *DCC 1999*, pages 316–325, 1999.

[51] P. Gawrychowski. Tying up the loose ends in fully LZW-compressed pattern matching. In *STACS 2012*, volume 14 of *LIPIcs*, pages 624–635, 2012.

[52] P. Gawrychowski and A. Jez. LZ77 factorisation of trees. In *FSTTCS 2016*, volume 65 of *LIPIcs*, pages 35:1–35:15, 2016.

[53] P. Gawrychowski, A. Karczmarz, T. Kociumaka, J. Lacki, and P. Sankowski. Optimal dynamic strings. In *SODA 2018*, pages 1509–1528. SIAM, 2018.

[54] S. Giuliani, S. Inenaga, Z. Lipták, N. Prezza, M. Sciortino, and A. Toffanello. Novel results on the number of runs of the Burrows-Wheeler-transform. In *SOFSEM*, pages 249–262, 2021.

[55] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[56] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

[57] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB Endow.*, 5(3):265–273, 2011.

[58] T. I. Longest common extensions with recompression. In *CPM 2017*, volume 78 of *LIPIcs*, pages 18:1–18:15, 2017.

[59] A. Jez. A really simple approximation of smallest grammar. *Theor. Comput. Sci.*, 616:141–150, 2016.

[60] D. Kempa and T. Kociumaka. Resolution of the Burrows-Wheeler transform conjecture. In *FOCS 2020*, pages 1002–1013. IEEE, 2020.

[61] D. Kempa and N. Prezza. At the roots of dictionary compression: string attractors. In *Proc. STOC*, pages 827–840, 2018.

[62] D. Kempa and B. Saha. An upper bound and linear-space queries on the lz-end parsing. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, pages 2847–2866. SIAM, 2022.

[63] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *DCC 1998*, pages 103–112. IEEE Computer Society, 1998.

[64] J. C. Kieffer and E. Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Trans. Information Theory*, 46(3):737–754, 2000.

[65] T. Kociumaka, G. Navarro, and F. Olivares. Near-optimal search time in δ-optimal space. *CoRR*, abs/2206.00781, 2022.

[66] T. Kociumaka, G. Navarro, and N. Prezza. Towards a definitive measure of repetitiveness. In *LATIN*, pages 207–219, 2020.

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

[68] G. Koulouras and M. C. Frith. Significant non-existence of sequences in genomes and proteomes. *Nucleic acids research*, 49(6):3139–3155, 2021.

[69] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013.

[70] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In E. Chávez and S. Lonardi, editors, *SPIRE 2010*, volume 6393 of *Lecture Notes in Computer Science*, pages 201–206, 2010.

[71] G. Lagarde and S. Perifel. Lempel-Ziv: a "one-bit catastrophe" but not a tragedy. In *SODA*, pages 1478–1495, 2018.

[72] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. DCC*, pages 296–305, 1999.

[73] M. Lohrey, S. Maneth, and R. Mennicke. XML tree structure compression using repair. *Inf. Syst.*, 38(8):1150–1167, 2013.

[74] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nord. J. Comput.*, 12(1):40–66, 2005.

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

[76] S. Mantaci, A. Restivo, G. Romana, G. Rosone, and M. Sciortino. A combinatorial view on string attractors. *Theor. Comput. Sci.*, 850:236–248, 2021.

[77] S. Maruyama, M. Nakahara, N. Kishiue, and H. Sakamoto. ESP-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013.

[78] K. Mehlhorn, R. Sundar, and C. Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[79] T. Mieno, Y. Kuhara, T. Akagi, Y. Fujishige, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *SOFSEM 2020*, volume 12011 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2020.

[80] S. Mitsuya, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Compressed communication complexity of hamming distance. *CoRR*, abs/2103.03468, 2021.

[81] G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004.

[82] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[83] G. Navarro. Document listing on repetitive collections with guaranteed performance. *Theor. Comput. Sci.*, 772:58–72, 2019.

[84] G. Navarro. Indexing highly repetitive string collections, part II: compressed indexes. *ACM Comput. Surv.*, 54(2):26:1–26:32, 2021.

[85] G. Navarro and N. Prezza. Universal compressed text indexing. *Theor. Comput. Sci.*, 762:41–50, 2019.

[86] G. Navarro and C. Urbina. On stricter reachable repetitiveness measures. In *SPIRE 2021*, volume 12944 of *Lecture Notes in Computer Science*, pages 193–206. Springer, 2021.

[87] G. Nelson, J. C. Kieffer, and P. C. Cosman. An interesting hierarchical lossless data compression algorithm, 1995. Invited Presentation.

[88] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Intell. Res.*, 7:67–82, 1997.

[89] T. Nishimoto, T. I, S. Inenaga, H. Bannai, and M. Takeda. Dynamic index and LZ factorization in compressed space. *Discret. Appl. Math.*, 274:116–129, 2020.

[90] T. Nishimoto and Y. Tabei. Optimal-time queries on BWT-runs compressed indexes. In *ICALP 2021*, volume 198 of *LIPIcs*, pages 101:1–101:15, 2021.

[91] T. Nishimoto and Y. Tabei. R-enum: Enumeration of characteristic substrings in BWT-runs bounded space. In *CPM 2021*, volume 191 of *LIPIcs*, pages 21:1–21:21, 2021.

[92] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Trans. Computers*, 60(10):1471–1484, 2011.

[93] D. S. Nunes, F. A. Louza, S. Gog, M. Ayala-Rincón, and G. Navarro. Grammar compression by induced suffix sorting. *ACM Journal of Experimental Algorithmics (JEA)*, 27:1–33, 2022.

[94] D. S. N. Nunes, F. A. da Louza, S. Gog, M. Ayala-Rincón, and G. Navarro. A grammar compression algorithm based on induced suffix sorting. In *Proc. DCC*, pages 42–51, 2018.

[95] D. Pratas and J. M. Silva. Persistent minimal sequences of sars-cov-2. *Bioinformatics*, 36(21):5129–5132, 2020.

[96] N. Prezza. On locating paths in compressed tries. In *SODA 2021*, pages 744–760. SIAM, 2021.

[97] S. J. Puglisi and B. Zhukova. Smaller RLZ-compressed suffix arrays. In *Proc. DCC*, pages 213–222, 2021.

[98] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[99] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *Proc. FOCS*, pages 320–328, 1996.

[100] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. SPIRE*, volume 5280 of *LNCS*, pages 164–175, 2008.

[101] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, 1982.

[102] 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.

[103] 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.

[104] Y. Tamakoshi, K. Goto, S. Inenaga, H. Bannai, and M. Takeda. An opportunistic text indexing structure based on run length encoding. In *CIAC 2015*, volume 9079 of *Lecture Notes in Computer Science*, pages 390–402. Springer, 2015.

[105] K. Tsuruta, D. Köppl, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda. Grammar-compressed self-index with Lyndon words. *IPSJ TOM*, 13(2):84–92, 2020.

[106] N. Varma and Y. Yoshida. Average sensitivity of graph algorithms. In *SODA 2021*, pages 684–703. SIAM, 2021.

[107] P. Weiner. Linear pattern matching algorithms. In *Proc. SWAT*, pages 1–11, 1973.

[108] E.-H. Yang and J. C. Kieffer. Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform. i. without context models. *IEEE Transactions on Information Theory*, 46(3):755–777, 2000.

[109] Y. Yoshida and S. Zhou. Sensitivity analysis of the maximum matching problem. In *ITCS 2021*, volume 185 of *LIPIcs*, pages 58:1–58:20, 2021.

[110] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, IT-23(3):337–349, 1977.

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