Proceedings of the Prague Stringology Conference 2026

Edited by Jan Holub and Jan Žďárek

August 2026

Prague Stringology Club
http://www.stringology.org/

ISBN 978-80-01-07576-0

Preface

The proceedings in your hands contain a collection of papers presented in the Prague Stringology Conference 2026 (PSC 2026) held on August 24–25, 2026 at the Czech Technical University in Prague, which organizes the event. The conference focused on stringology, i.e., a discipline concerned with algorithmic processing of strings and sequences and related topics.

The submitted papers were reviewed by the program committee subject to originality and quality. The eight papers in this proceedings made the cut and were selected for regular presentation at the conference.

The PSC 2026 was organized in both present and remote form. Speakers we required to present their papers in person. Non-speakers could decide whether to arrive to Prague or to participate remotely.

The Prague Stringology Conference has a long tradition. PSC 2026 is the twenty-ninth PSC conference. In the years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conferences (PSC's) in 2001–2006, 2008–2021, 2023–25 preceded this conference. The proceedings of these workshops and conferences have been published by the Czech Technical University in Prague and are available on the web pages of the Prague Stringology Club. Selected contributions have been regularly published in special issues of journals such as: Kybernetika, the Nordic Journal of Computing, the Journal of Automata, Languages and Combinatorics, the International Journal of Foundations of Computer Science, and the Discrete Applied Mathematics.

The Prague Stringology Club was founded in 1996 as a research group at the Czech Technical University in Prague. The goal of the Prague Stringology Club is to study algorithms on strings, sequences, and trees with an emphasis on automata theory. The first event organized by the Prague Stringology Club was the workshop PSCW'96 featuring only a handful of invited talks. However, since PSCW'97 the papers and talks are selected by a rigorous peer review process. The objective is not only to present new results in stringology and related areas but also to facilitate personal contacts among the people working on these problems.

We would like to thank all those who had submitted papers for PSC 2026 as well as the reviewers. Special thanks go to all the members of the program committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC 2026. Last but not least, our thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Prague, Czech Republic
on August 2026

Jan Holub and Ferdinando Cicalese

Conference Organisation

Program Committee

Amihood Amir	(Bar-Ilan University, Israel)
Ferdinando Cicalese, *Chair*	(University of Verona, Italy)
Nicola Cotumaccio	(University of Helsinki, Finland)
Simone Faro	(Università di Catania, Italy)
Pamela Fleishmann	(Kiel University, Germany)
Pawel Gawrychowski	(University of Wroclaw, Poland)
Jan Holub, *Chair*	(Czech Technical University in Prague, Czech Republic)
Shmuel T. Klein	(Bar-Ilan University, Israel)
Dominik Köppl	(Tokyo Medical and Dental University, Japan)
Thierry Lecroq	(Université de Rouen, France)
Robert Mercas	(Loughborough University, United Kingdom)
Yuto Nakashima	(Kyushu University, Japan)
Solon Pissis	(CWI, The Netherlands)
Giulia Punzi	(University of Pisa, Italy)

Organising Committee

Dominika Bohuslavová	Josef Erik Sedláček	Jan Trávníček
Jan Holub, *Chair*	Jana Šmídová	Jan Žďárek
Tomáš Pecka		

External Referees

Frank Drewes	Arnaud Lefebvre	Hilde Verbeek
Jonas Ellert	Francesco Pio Marino	Max Wiedenhöft
Annika Huch	Yoshifumi Sakai	

Table of Contents

Invited Talk

Pattern Matching Meets Privacy: Algorithms and Data Structures for Analyzing Sensitive String Data *by Teresa Anna Steiner* 1

Contributed Talks

In-Place BWT and Lyndon Array Construction in Constant Space *by Felipe A. Louza and Arnaud Lefebvre* 3

Sliding Suffix Trees Revisited *by Laurentius Leonard, Shunsuke Inenaga, Hideo Bannai, and Takuya Mieno* 13

Faster Indeterminate Pattern Matching Algorithm with Practical Implementations *by Hamed Hasibi, Neerja Mhaskar, and Tieyun Zhang* 28

On the Additive Sensitivity of CDAWG-grammars *by Hiroto Fujimaru and Shunsuke Inenaga* 43

Relaxation of Square-Freeness *by Hiroki Shibata, Takuya Mieno, Dominik Köppl, and Shunsuke Inenaga* 58

A Lower-Bound Cascade for Lossless Pruning in Levenshtein-Based Contour Matching *by Stanisław Piotr Skulimowski* 71

Computation of Periods and Covers Using Sampling *by Thierry Lecroq, Arnaud Lefebvre, and Francesco Pio Marino* 85

Updating Permutation-Compressed Hypergraphs *by Enno Adler, Stefan Böttcher, and Rita Hartel* 97

Author Index 113

Pattern Matching Meets Privacy: Algorithms and Data Structures for Analyzing Sensitive String Data

(*Abstract*)

Teresa Anna Steiner

Department of Mathematics and Computer Science
Faculty of Science
University of Southern Denmark
Steiner@imada.sdu.dk

Analyzing patterns within collections of strings is extremely useful in a wide range of applications, like DNA sequence analysis and finding frequent patterns in transit data. However, the results of these analysis tasks may expose confidential information about individuals. Differential privacy is a framework which has successfully been employed to give strong privacy guarantees to data analysis tasks. It uses carefully calibrated random noise to hide the influence of any individual on the output of a given query. The challenge in differential privacy research is to design algorithms which fulfill this privacy guarantee and give accurate results at the same time.

In this talk, I will give a short introduction to differential privacy and show how its framework can be applied to strings and pattern matching. We will see which combinatorial properties of data type and query are relevant when hoping to design algorithms which are both private and accurate, and introduce useful tools in this context. Combining these tools with ideas from data structure design, we give a differentially private data structure for counting patterns in a collection of documents with almost optimal error. This is based on joint work with Giulia Bernardini, Philip Bille, and Inge Li Gørtz.

Teresa Anna Steiner: Pattern Matching Meets Privacy: Algorithms and Data Structures for Analyzing Sensitive String Data, p. 1.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

In-Place BWT and Lyndon Array Construction in Constant Space

Felipe A. Louza[1,2] and Arnaud Lefebvre[3⋆]

[1] Universidade Federal de Uberlândia, Uberlândia, MG, Brazil
[2] Universidade Estadual de Campinas, Campinas, SP, Brazil
louza@ufu.br
[3] Normandie Univ., 76000 Rouen, France
arnaud.lefebvre@univ-rouen.fr

Abstract. We show that the BWT and the Lyndon array of a string can be constructed simultaneously using only $O(1)$ additional memory words in the general ordered alphabet model. Our algorithm extends the in-place quadratic-time BWT construction of Crochemore *et al.* [10] by maintaining suffix ranks during the right-to-left construction, from which the Lyndon array is derived via a next-smaller-value procedure. We also consider the construction of the succinct $2n$-bit representation of the Lyndon array under the same constant-space constraints. Although not practical, our methods demonstrate that these fundamental structures admit in-place constructions under the strongest possible space bounds, even over unbounded alphabets.

Keywords: BWT, Lyndon Array, Succinct data structures, In-place algorithms, String processing

1 Introduction

The Burrows–Wheeler Transform (BWT) [8] and the Lyndon array (LA) [14] are fundamental structures in string processing. The BWT underlies compressed indexing [13] and data compression [8], while the LA captures the combinatorial structure of a string through its Lyndon factorization and plays a role in several pattern-matching [5,4] and combinatorial algorithms [2,3]

The BWT depends on the lexicographic ordering of all suffixes of the input string. A standard approach computes the BWT in $O(n)$ time after constructing the suffix array (SA) [19], which requires $O(n \log n)$ bits of extra space. Several algorithms compute the BWT directly in linear time using only $O(n \log \sigma)$ bits of working space, including the induced-sorting algorithm of Okanohara and Sadakane [23] and subsequent deterministic constructions [20,6]. Under the much stronger restriction of using only $O(1)$ additional words, however, the most efficient known algorithm is the in-place $O(n^2)$ construction of Crochemore *et al.* [10], which overwrites the input text while requiring only constant extra memory. Their method works in the general ordered alphabet model, requiring only constant-time symbol comparisons and no assumptions on the alphabet size.

The Lyndon array can likewise be computed from the suffix ordering in $O(n)$ time using $O(n \log n)$ bits of extra space [14,15,17,22]. Alternatively, it can be obtained via Duval's factorization algorithm [12] in $O(n^2)$ time using $O(1)$ extra space. More recently, Bille *et al.* [7] showed that the Lyndon array can be computed in $O(n)$ time using only $O(1)$ extra words, and in $o(n)$ extra bits for the succinct representation [18]. Their algorithm, like that of Crochemore *et al.*, operates in the general ordered alphabet model and does not depend on σ.

⋆ Corresponding author

Felipe A. Louza, Arnaud Lefebvre: In-Place BWT and Lyndon Array Construction in Constant Space, pp. 3–12.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

i	Sorted suffix $T_{\mathsf{SA}[i]}$	$\mathsf{SA}[i]$	$\mathsf{ISA}[i]$	$\mathsf{BWT}[i]$	$\mathsf{LA}[i]$
0	\$	6	4	A	1
1	A\$	5	3	N	2
2	ANA\$	3	6	N	1
3	ANANA\$	1	2	B	2
4	BANANA\$	0	5	\$	1
5	NA\$	4	1	A	1
6	NANA\$	2	0	A	1

$$\mathsf{LA}_{\mathsf{BP}} = ()(())(())()()$$

Figure 1. SA, ISA, BWT and LA for $T = \texttt{BANANA\$}$. The Lyndon array $\mathsf{LA}[i]$ gives the length of the longest Lyndon word starting at position i. $\mathsf{LA}_{\mathsf{BP}}$ gives the succinct $2n$-bit representation of LA.

Our contribution. In this paper, we extend the quadratic in-place BWT algorithm of Crochemore *et al.* [10] to *simultaneously* compute the Lyndon array, while still using only $O(1)$ extra space. The resulting algorithm constructs both structures in $O(n^2)$ time using constant additional space. Compared with the linear-time algorithm of Bille *et al.* [7], our approach is asymptotically slower but computes the BWT *in place* while simultaneously constructing the Lyndon array using only $O(1)$ additional space. We further show that the succinct $2n$-bit representation of the Lyndon array can be constructed under the same constant-space constraints in $O(n^{3+\epsilon})$ time. In contrast, the construction of Bille *et al.* [7] achieves linear time by allowing additional $o(n)$ bits of working space. Alternatively, the succinct Lyndon array can be computed in $O(n/\log_\sigma n)$ time using extra space, as shown in [1].

2 Preliminaries

Let Σ be an ordered alphabet of σ symbols, which may be unbounded. We assume symbols in Σ are compared in constant-time. Let \$ be a special symbol (end marker) not in Σ that is lexicographically smaller than every symbol in Σ. We denote by Σ^* the set of all finite nonempty strings over Σ, and we define $\Sigma^*\$ = \{\, T\$ \mid T \in \Sigma^* \,\}$.

For a string $T = T[0]T[1]\dots T[n-1]$ over $\Sigma^*\$$, of length n, we write $T[i,j] = T[i]T[i+1]\cdots T[j]$ for the substring starting at position i and ending at j, where $0 \le i \le j < n$. A *prefix* of T is a substring of the form $T[0,k]$, and a *suffix* is a substring of the form $T[k,n-1]$, with $0 \le k < n$. We denote the suffix that begins at position i by T_i.

The *suffix array* (SA) [19] of $T \in \Sigma^*\$$ is an array $\mathsf{SA}[0,n-1]$ that contains a permutation of $\{0,\dots,n-1\}$ such that

$$T_{\mathsf{SA}[0]} < T_{\mathsf{SA}[1]} < \cdots < T_{\mathsf{SA}[n-1]}$$

in lexicographic order. The *inverse suffix array* (ISA) is the inverse permutation defined by $\mathsf{ISA}[\mathsf{SA}[i]] = i$. Hence, $\mathsf{ISA}[k]$ gives the lexicographic rank of the suffix T_k.

The *Burrows–Wheeler Transform* (BWT) [8] of a string T is a permutation of its characters that tends to cluster equal symbols. It can be defined using the suffix

array as

$$\mathsf{BWT}[i] = \begin{cases} T[\mathsf{SA}[i]-1], & \text{if } \mathsf{SA}[i] > 0, \\ \$, & \text{otherwise.} \end{cases}$$

A *Lyndon word* [9] is a nonempty string that is strictly smaller in lexicographic order than all of its proper nonempty suffixes. Every string can be uniquely factorized as a concatenation of Lyndon words in nonincreasing lexicographic order (the *Lyndon factorization*).

The *Lyndon array* (LA) [14] (also called Lyndon table [11]) of a string $T \in \Sigma^*\$$ is an array $\mathsf{LA}[0, n-1]$ that stores, for each position i, the length of the longest Lyndon word starting at i. Formally,

$$\mathsf{LA}[i] = \min\{\, j - i \mid j > i,\ T[i, j-1] \text{ is not smaller than } T[j, n-1] \,\},$$

or equivalently, the Lyndon array can be computed from the inverse suffix array using the *next smaller value* (NSV) function:

$$\mathsf{LA}[i] = \mathrm{NSV}_{\mathsf{ISA}}(i) - i, \tag{1}$$

where

$$\mathrm{NSV}_{\mathsf{ISA}}(i) = \min\{\, j > i \mid \mathsf{ISA}[j] < \mathsf{ISA}[i] \,\}, \tag{2}$$

and $\mathrm{NSV}_{\mathsf{ISA}}(i) = n$ if no such j exists.

Figure 1 shows the SA, ISA, BWT and Lyndon array for $T = \texttt{BANANA\$}$.

The Lyndon array can be represented as a succinct balanced-parentheses sequence (BPS) [21] that requires only $2n$ bits of space. Introduced in [18], when augmented with $o(n)$ additional bits, this representation allows every value $\mathsf{LA}[i]$ to be retrieved in $O(1)$.

Let T be a string of length n and let $\mathsf{LA}[0..n-1]$ be its Lyndon array. Define an array $\mathsf{R}[0..n-1]$ by

$$\mathsf{R}[i] = i + \mathsf{LA}[i]. \tag{3}$$

The array R satisfies $\mathsf{R}[i] > i$ for all i and induces a set of properly nested intervals $[i, \mathsf{R}[i])$, which induces an ordered tree, called *Lyndon tree* [11], whose structure can be encoded as BPS.

The BPS encoding of the nested intervals, stored in $\mathsf{LA}_{\mathsf{BP}}[0, 2n-1]$, is computed by scanning positions $i = 0, 1, \ldots, n-1$ of LA. Let pos denote the index of the next free position in $\mathsf{LA}_{\mathsf{BP}}$, initially set to 0. When processing a position i, an opening parenthesis "(" corresponding to i is written at position $\mathsf{LA}_{\mathsf{BP}}[\mathsf{pos}]$. The matching closing parenthesis ")" is written at position $\mathsf{LA}_{\mathsf{BP}}[\mathsf{pos} + 2 \cdot \mathsf{LA}[i] - 1]$. Afterwards, pos is advanced to the smallest index such that $\mathsf{LA}_{\mathsf{BP}}[\mathsf{pos}]$ is still undefined. Because each position of LA contributes exactly one opening and one matching closing parenthesis, the resulting BPS contains n matching pairs and therefore occupies exactly $2n$ bits.

Given the BPS $\mathsf{LA}_{\mathsf{BP}}[0, 2n-1]$. Let $\mathsf{open}(i)$ (resp. $\mathsf{close}(i)$) denote the position of the i-th opening (resp. closing) parenthesis in $\mathsf{LA}_{\mathsf{BP}}$. By construction, we have

$$\mathsf{LA}[i] = \frac{\mathsf{close}(i) - \mathsf{open}(i) + 1}{2} \tag{4}$$

Using standard balanced-parentheses navigation operations, the position of $\mathsf{open}(i)$ and $\mathsf{close}(i)$ can be computed in $O(1)$ time when $\mathsf{LA}_{\mathsf{BP}}$ is augmented with $o(n)$ additional bits, and thus every value $\mathsf{LA}[i]$ can be retrieved in constant time (see [18] for details).

	s	BWT	sorted suffixes
$s \rightarrow$	0	?	BANANA$
	1	A	$
	2	N	A$
	3	N	ANA$
$p \rightarrow$	4	$	ANANA$
	5	A	NA$
	6	A	NANA$

	s	BWT	sorted suffixes
	0	A	$
	1	N	A$
	2	N	ANA$
	3	B	ANANA$
$r \rightarrow$	4	$	BANANA$
	5	A	NA$
	6	A	NANA$

Figure 2. After Step 1 (left) and Step 4 (right) for $T =$ `BANANA$`, $s = 0$.

Figure 1 also shows the succinct $2n$-bit representation of the Lyndon array for $T =$ `BANANA$`.

3 In-Place BWT

The in-place BWT algorithm of Crochemore *et al.* [10] computes the transform in quadratic time using only constant extra space, overwriting the input string from right to left with its BWT as the construction proceeds.

The BWT is computed inductively on the length of the suffix. Let $BWT(T_s)$ denote the BWT of the suffix $T_s = T[s, n-1]$. The base cases are the two rightmost suffixes:

$$BWT(T_{n-2}) = T_{n-2}, \quad BWT(T_{n-1}) = T_{n-1}.$$

For the inductive step, assume that $BWT(T_{s+1})$ is already stored in $T[s+1, n-1]$. The algorithm computes $BWT(T_s)$ by inserting the new suffix T_s in its correct lexicographic position among the suffixes $T_{s+1}, \ldots, T_{n-1}$. It relies on the fact that the position of the end marker \$ in $BWT(T_{s+1})$ corresponds to the rank of the suffix T_{s+1} among the current suffixes.

The algorithm consists of four steps, for iterations $s = n-2, n-3, \ldots, 0$.

1. **Find the position of the end marker.** Locate the position p of \$ in $T[s+1, n-1]$.
2. **Compute the rank of the new suffix.** Let $c = T[s]$ be the first symbol of T_s. The local rank r of T_s among the suffixes $T_s, \ldots, T_{n-1}$ is obtained by summing the number of symbols in $T[s+1, n-1]$ that are strictly smaller than c, with the number of occurrences of c in $T[s+1, p]$.
3. **Insert T_s into the sequence.** Store c into $T[p]$, replacing the end marker \$.
4. **Shift symbols to insert a new end marker.** Shift the block $T[s+1, r]$ one position to the left and write \$ at position $T[r]$. Then, $BWT(T_s) = T_s$.

Figure 2 illustrates the execution of the in-place BWT algorithm on the text `BANANA$` for the iteration $s = 0$, showing the effect of Steps 1 and 4 on the partial $BWT(T_1)$ stored in $T[1\,..\,n-1]$.

The algorithm runs in $O(n^2)$ time and uses only $O(1)$ additional variables. Its simplicity and elegant inductive structure make it a useful foundation for extensions, such as computing the LCP array [16] or, as we show next, the inverse suffix array and then the Lyndon array, while preserving the constant-space property.

	s	BWT	ISA	sorted suffixes
$s \rightarrow$	0	?	?	BANANA$
	1	A	3	$
	2	N	5	A$
	3	N	2	ANA$
$p \rightarrow$	4	$	4	ANANA$
	5	A	1	NA$
	6	A	0	NANA$

	s	BWT	ISA	sorted suffixes
	0	A	4	$
	1	N	3	A$
	2	N	6	ANA$
	3	B	2	ANANA$
$r \rightarrow$	4	$	5	BANANA$
	5	A	1	NA$
	6	A	0	NANA$

Figure 3. After Step 1 (left) and Step 5 (right) for $T =$ `BANANA$`, $s = 0$.

4 In-place BWT along with the Lyndon Array

We now describe how to extend the in-place BWT algorithm (Section 3) to compute the BWT and the Lyndon array (LA) simultaneously, using only constant additional memory.

The key idea is to maintain, during each iteration $s = n-2, n-3, \ldots, 0$, the lexicographic ranks among the suffixes $T_s, \ldots, T_{n-1}$, stored in the $\mathsf{ISA}[s, n-1]$. At the end, the values in $\mathsf{ISA}[0, n-1]$ are converted into the Lyndon array, according to Equations 1 and 2.

The algorithm proceeds inductively on the length of the suffix, as in the original in-place BWT construction. Let $\mathsf{ISA}(T_s)$ denote the partial ISA for $T_s = T[s, n-1]$. The base cases for ISA are:

$$\mathsf{ISA}(T_{n-2}) = [1, 0], \quad \mathsf{ISA}(T_{n-1}) = [0].$$

For the inductive step, assume that we also have $\mathsf{ISA}(T_{s+1})$ already stored in $\mathsf{ISA}[s+1, n-1]$. The algorithm computes $\mathsf{ISA}(T_s)$ by storing in $\mathsf{ISA}[s]$ the rank r of the new suffix T_s inserted into $BWT(T_{s+1})$, and by updating the ISA-values of all suffix that has rank larger than r in $\mathsf{ISA}[s+1, n-1]$. Therefore, our algorithm has a new Step 5, added just after Step 4 of the in-place BWT algorithm as follows.

5. **Update the inverse suffix array.** After inserting T_s in $BWT(T_{s+1})$, set $\mathsf{ISA}[s] = r$, and increment by one all entries $\mathsf{ISA}[i]$, with $s+1 \leq i < n$, such that $\mathsf{ISA}[i] \geq r$.

Lemma 1. *Step 5 maintains the correct rank information for all suffixes currently present in $BWT(T_s)$.*

Proof. We prove the invariant by induction on s (right to left). For the base $s = n-2$ the two ranks are set correctly. Assume that before processing s the array $\mathsf{ISA}[s+1..n-1]$ stores the ranks $0, \ldots, m-1$ of $T_{s+1}, \ldots, T_{n-1}$, with $m = n-s$. Let $r \in \{0, \ldots, m\}$ be the rank of T_s among $T_s, \ldots, T_{n-1}$. Every suffix with old rank $q < r$ are shifted one position to the left and keep their rank, while those with $q \geq r$ must have its rank increased by 1, as T_s was inserted into position r; moreover $\mathsf{ISA}[s]$ must be r. Step 5 performs exactly these updates (incrementing entries with $\mathsf{ISA}[i] \geq r$ and setting $\mathsf{ISA}[s] = r$). Hence after the update $\mathsf{ISA}[s..n-1]$ stores the ranks $0, \ldots, m$ of $T_s, \ldots, T_{n-1}$, which proves the claim.

Figure 3 illustrates the execution of the algorithm on the text `BANANA$` for the iteration $s = 0$, showing the effect of Steps 1 and 5 on the partial BWT and ISA.

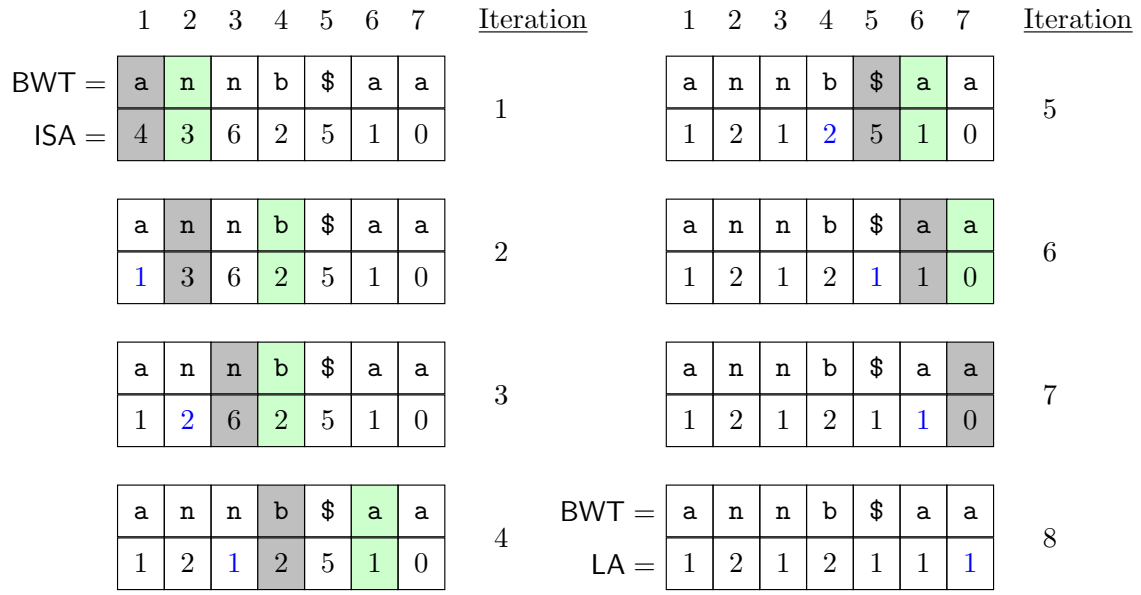

Figure 4. Example of the in-place LA construction from ISA on $T =$ `BANANA$`.

Finally, given the inverse suffix array $\mathsf{ISA}(T_0)$, we scan $\mathsf{ISA}[0, n-1]$ from left to right, overwriting each processed entry with the corresponding Lyndon array value. For each position $i = \{0, 1, \ldots, n-1\}$. Then, $\mathsf{LA}[i] = j - i$. This corresponds to computing the next smaller value (NSV) over the ISA, as described in Section 2. Since $\mathsf{ISA}[i]$ is no longer needed after processing position i, it can be overwritten by $\mathsf{LA}[i]$, yielding the final Lyndon array without allocating any additional memory.

Figure 4 illustrates how the ISA values maintained during the in-place BWT construction are subsequently overwritten by the corresponding Lyndon array values.

In the appendix, we provide a C implementation of the algorithm, including the in-place construction of the BWT (Lines 10–28), the maintenance of the inverse suffix array during the construction process (Lines 30–34), and the subsequent derivation of the Lyndon array from the computed ranks using the next-smaller-value procedure (Lines 36–43).

Theorem 2. *The BWT can be computed in-place together with Lyndon array in $O(n^2)$ time.*

Proof. The algorithm follows the same inductive structure as the in-place BWT construction. Each iteration performs a constant amount of work plus two $O(n)$ scans (Steps 1–2), and there are n iterations, yielding an overall running time of $O(n^2)$. The update of the inverse suffix array in Step 5 requires only an extra linear scan at each iteration, and the final conversion of ISA into the Lyndon array by the next-smaller-value scan also takes $O(n^2)$ time. Thus, the total time remains quadratic. The space usage is $O(1)$ words beyond the input text T and the output array LA.

5 Computing the Succinct Lyndon Array

The succinct $2n$-bit representation of the Lyndon array [18] can be computed directly from the BWT of T, using $O(1)$ additional working space. Our construction follows the in-place BWT inversion algorithm [10, Section 3]

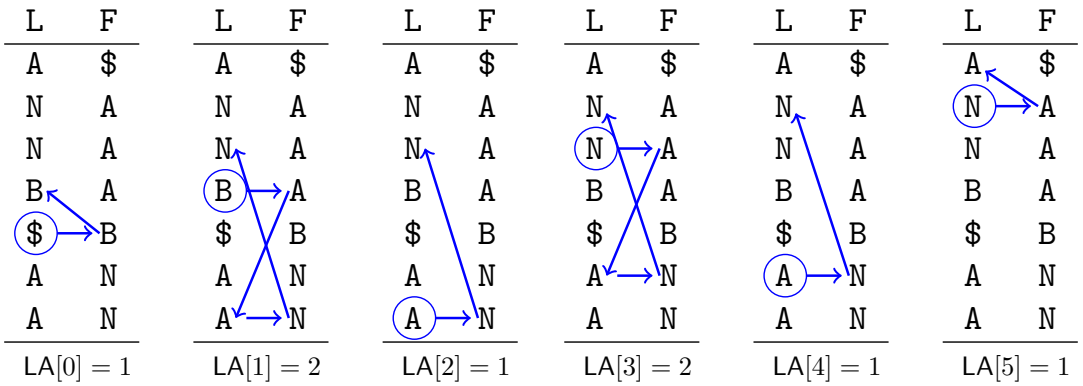

Figure 5. Computing the LA-values from the BWT of T.

Initially, $\mathsf{BWT}(T)$ is stored in the array $L[0..n-1]$, while the balanced-parentheses representation $\mathsf{LA_{BP}}[0..2n-1]$ is empty. We begin by scanning L to locate the position p of the symbol \$, which corresponds to the value $\mathsf{ISA}[0]$. For each step $i = 0, 1, \ldots, n-2$, let $p = \mathsf{ISA}[i]$. Using the in-place BWT inversion procedure, we iteratively apply the LF-mapping [8] starting from position p in order to compute the positions

$$\mathsf{ISA}[i+1],\ \mathsf{ISA}[i+2],\ \ldots,\ \mathsf{ISA}[i+j],$$

which correspond to the rank of $T_{i+1}, T_{i+2}, \ldots, T_{i+j}$. This procedure continues until the first position $q = \mathsf{ISA}[i+j]$ such that $q < p$ is encountered. By Equation 2, this index j equals $\mathrm{NSV}_{\mathsf{ISA}}(i)$, and therefore

$$\mathsf{LA}[i] = j - i.$$

This value is encoded directly into $\mathsf{LA_{BP}}[0..2n-1]$ by writing an opening parenthesis "(" at the current leftmost free position pos, and the matching closing parenthesis ")" at position $\mathsf{pos} + 2\mathsf{LA}[i] - 1$, as described in Section 2. The pointer pos is then advanced to the next free position, and the algorithm proceeds with step $i+1$. For the last position $i = n-1$, since $T[n-1] = \$$, the suffix T_{n-1} is lexicographically smaller than all other suffixes and therefore $\mathsf{LA}[n-1] = 1$, which is encoded directly in $\mathsf{LA_{BP}}$.

Figure 5 illustrates this process for $T = \texttt{BANANA\$}$. Starting from the position of the end marker, successive LF-mapping steps identify the first suffix whose lexicographic rank is smaller than the current one.

Theorem 3. *The succinct $2n$-bit Lyndon array can be computed in-place in $O(n^{3+\epsilon})$ time.*

Proof. Given $\mathsf{BWT}(T)$, the algorithm processes each position $i = 0, 1, \ldots, n-1$ by performing successive LF-mapping steps until the value $\mathsf{LA}[i]$ is determined. In the worst case, this requires $O(n)$ LF steps per position. Each LF step is computed using the in-place BWT inversion algorithm of Crochemore *et al.* [10], whose running time is $O(n^{1+\varepsilon})$ in the general ordered alphabet model, for any fixed constant $\varepsilon > 0$. Therefore, the total running time is $O(n^{3+\varepsilon})$. The construction maintains only a constant number of counters and indices in addition to the input and output arrays, and thus uses $O(1)$ extra space.

6 Conclusion

We have presented a constant-space algorithm that computes the BWT and the Lyndon array simultaneously. Our method extends the in-place BWT construction of Crochemore *et al.* [10] by maintaining the inverse suffix array incrementally and then deriving the Lyndon array. Although not intended for practical use, these results show constructions for these fundamental structures under the strongest space bounds. We also showed that the succinct $2n$-bit representation of the Lyndon array can be obtained directly from the BWT using constant space. Unlike the linear-time construction that relies on additional working memory [18,7], our procedure requires only constant extra space, at the cost of higher time complexity.

Acknowledgments

The first author was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), grants 408314/2023-0, 311128/2025-4 and 351599/2025-8, and by the São Paulo Research Foundation (FAPESP), process 2025/16208-7.

References

1. H. Bannai and J. Ellert: *Lyndon arrays in sublinear time*, in Proc. European Symposium on Algorithms, 2023, pp. 14:1–14:16.
2. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *A new characterization of maximal repetitions by Lyndon trees.* CoRR, abs/1406.0263 2014.
3. H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta: *The "runs" theorem.* SIAM J. Comput., 46(5) 2017, pp. 1501–1514.
4. H. Bannai, J. Kärkkäinen, D. Köppl, and M. Piatkowski: *Constructing and indexing the bijective and extended Burrows-Wheeler transform.* Inf. Comput., 297 2024, p. 105153.
5. H. Bannai, D. Köppl, and Z. Lipták: *A survey of the bijective Burrows-Wheeler transform*, in The Expanding World of Compressed Data: A Festschrift for Giovanni Manzini's 60th Birthday, Manzini's Festschrift, Venice, Italy, July 25, 2025, P. Ferragina, T. Gagie, and G. Navarro, eds., vol. 131 of OASIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 2:1–2:26.
6. D. Belazzougui, F. Cunial, J. Kärkkäinen, and V. Mäkinen: *Linear-time string indexing and analysis in small space.* ACM Trans. Algorithms, 16(2) 2020, pp. 17:1–17:54.
7. P. Bille, J. Ellert, J. Fischer, I. L. Gørtz, F. Kurpicz, J. I. Munro, and E. Rotenberg: *Space efficient construction of Lyndon arrays in linear time*, in Proc. International Colloquium on Automata, Languages, and Programming, 2020, pp. 14:1–14:18.
8. M. Burrows and D. Wheeler: *A block-sorting lossless data compression algorithm*, tech. rep., Digital SRC Research Report, 1994.
9. K. T. Chen, R. H. Fox, and R. C. Lyndon: *Free differential calculus IV — the quotient groups of the lower central series.* Ann. Math., 68 1958, pp. 81–95.
10. M. Crochemore, R. Grossi, J. Kärkkäinen, and G. M. Landau: *Computing the Burrows-Wheeler transform in place and in small space.* J. Discret. Algorithms, 32 2015, pp. 44–52.
11. M. Crochemore and L. M. S. Russo: *Cartesian and lyndon trees.* Theor. Comput. Sci., 806 2020, pp. 1–9.
12. J. Duval: *Factorizing words over an ordered alphabet.* J. Algorithms, 4(4) 1983, pp. 363–381.
13. P. Ferragina and G. Manzini: *Opportunistic data structures with applications*, in Proc. Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 2000, pp. 390–398.
14. F. Franek, A. S. M. S. Islam, M. S. Rahman, and W. F. Smyth: *Algorithms to compute the Lyndon array*, in Proc. Prague Stringology Conference, 2016, pp. 172–184.
15. F. Franek, A. Paracha, and W. F. Smyth: *The linear equivalence of the suffix array and the partially sorted lyndon array*, in Proc. Prague Stringology Conference, 2017, pp. 77–84.

16. F. A. Louza, T. Gagie, and G. P. Telles: *Burrows-Wheeler transform and LCP array construction in constant space.* J. Discret. Algorithms, 42 2017, pp. 14–22.
17. F. A. Louza, S. Mantaci, G. Manzini, M. Sciortino, and G. P. Telles: *Inducing the Lyndon array*, in Proc. International Symposium on String Processing and Information Retrieval (SPIRE), 2019, pp. 138–151.
18. F. A. Louza, W. Smyth, G. Manzini, and G. P. Telles: *Lyndon array construction during Burrows–Wheeler inversion.* Journal of Discrete Algorithms, 50 2018, pp. 2–9.
19. U. Manber and E. W. Myers: *Suffix arrays: A new method for on-line string searches.* SIAM J. Comput., 22(5) 1993, pp. 935–948.
20. J. I. Munro, G. Navarro, and Y. Nekrich: *Space-efficient construction of compressed indexes in deterministic linear time*, in Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19, P. N. Klein, ed., SIAM, 2017, pp. 408–424.
21. J. I. Munro and V. Raman: *Succinct representation of balanced parentheses and static trees.* SIAM J. Comput., 31(3) 2001, pp. 762–776.
22. D. S. Nunes, F. A. Louza, and G. P. Telles: *Space-efficient lyndon array construction from compressed texts.* Procedia Computer Science, 273 2025, pp. 413–420, XIII Latin American Algorithms, Graphs, and Optimization Symposium (LAGOS 2025).
23. D. Okanohara and K. Sadakane: *A linear-time Burrows-Wheeler transform using induced sorting*, in Proc. International Symposium on String Processing and Information Retrieval (SPIRE), 2009, pp. 90–101.

A Source Code

```
void bwt_lyndon_inplace(unsigned char *T, int n, int* LA){
  int i, j, p, r=1, s;
  unsigned char c;

  // base cases
  LA[n-2] = n-1;
  LA[n-1] = n-2;

  for (s=n-3; s>=0; s--) {
     c = T[s];

     /*steps 1 and 2*/
     r = s;
     for (i=s+1, T[i]!=END_MARKER; i++)
       if(T[i]<=c) r++;
     p = i;
     while (i<n)
       if (T[i++]<c) r++;

     /*step 3*/
     T[p]=c;

     /*step 4*/
     for (i=s; i<r; i++) {
       T[i]=T[i+1];
     }

     T[r]=END_MARKER;

     /*step 5*/
     for (i=s+1; i<n; i++) {
       if(LA[i]<=r) LA[i]--;
     }
     LA[s]= r;
  }
  /*ISA  to LA*/
  for (i=0; i<n; i++) {
    for(j=i+1; j<n; j++){
      if(LA[j]<LA[i])
        break;
    }
    LA[i]=j-i;
  }
}

```

Sliding Suffix Trees Revisited

Laurentius Leonard[1], Shunsuke Inenaga[2], Hideo Bannai[3], and Takuya Mieno[4]

[1] Department of Informatics Science and Technology, Kyushu University, Japan.
laurentiusleonard@outlook.com
[2] Department of Informatics, Kyushu University, Japan.
inenaga.shunsuke.380@m.kyushu-u.ac.jp
[3] M&D Data Science Center, Institute of Integrated Research, Institute of Science Tokyo, Japan.
hdbn.dsc@tmd.ac.jp
[4] Graduate School of Informatics and Engineering, University of Electro-Communications, Japan.
tmieno@uec.ac.jp

Abstract. The *sliding suffix tree* (Fiala & Greene, 1989) is a suffix tree that is maintained for a sliding window $W_i = T[i..i+d-1]$ of size d that shifts over an input text T of length n from left to right, for increasing $i = 1, \ldots, n-d+1$. It is known that the sliding suffix tree can be maintained in $O(n \log \sigma)$ time with $O(d)$ space, where σ is the alphabet size. Updating the sliding suffix tree from $W_i = T[i..i+d-1]$ to $W_{i+1} = T[i+1..i+d]$ requires the following three major tasks: (1) Delete the leaf that represents the longest suffix W_i, (2) Insert new leaves that represent the suffixes of W_{i+1} that appear exactly once in W_{i+1}, and (3) After the leaf deletion due to Task (1) and each leaf insertion due to Task (2), maintain the label $\langle \ell, r \rangle$ of every edge as a *valid* pair in the new window W_{i+1}, such that $i+1 \leq \ell \leq r \leq i+d$. In this paper, we present the first algorithm that performs Task (3) in $O(1)$ worst-case time per node deletion/insertion, which leads to another alternative to efficient sliding suffix tree construction. This is an improvement over the existing algorithms by Larsson (1996, 1999) and by Senft (2005) both of which can only perform Task (3) in $O(1)$ *amortized* time. Our key data structure is a non-trivial extension of *leaf pointers*, which were originally proposed by Brodnik and Jekovec (2018) for pattern matching with sliding suffix trees.

Keywords: suffix trees, sliding windows, online algorithms, string data structures

1 Introduction

The suffix tree [11] of a string T, denoted $\mathsf{STree}(T)$, is an edge-labeled rooted tree that represents all suffixes of T. It is well known that $\mathsf{STree}(T)$ can be stored in $O(n)$ space for any string T of length n by representing each edge label x with a pair $\langle i, j \rangle$ of positions in T such that $x = T[i..j]$. A suffix tree is a powerful string indexing data structure with a myriad of applications [4].

Among the variations of the suffix tree is the sliding suffix tree, proposed by Fiala and Greene [3]. Let $W_i = T[i..i+d-1]$ be a *sliding window* of (fixed) size d over a text T of length n. The sliding suffix tree is the suffix tree of sliding window W_i that is maintained and updated for increasing $i = 1, \ldots, n-d+1$. Sliding suffix trees are core data structures in data compression [3,5,6] including the original variant of the Lempel-Ziv 77 compression [12], as well as pattern matching [1], computing maximal absent words (MAWs) [2], minimal unique substrings (MUSs) [7], and string net frequencies (NFs) [8], in the sliding window model.

Shifting the current window $W_i = T[i..i+d-1]$ to the next window $W_{i+1} = T[i+1..i+d]$ involves deleting the leftmost character $T[i]$ and appending a new character $T[i+d]$ to the window. Thus, updating $\mathsf{STree}(W_i)$ to $\mathsf{STree}(W_{i+1})$ requires the following major tasks in the data structure:

Laurentius Leonard, Shunsuke Inenaga, Hideo Bannai, Takuya Mieno: Sliding Suffix Trees Revisited, pp. 13–27.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

(1) Delete the leaf that represents the longest suffix $W_i = T[i..i+d-1]$.
(2) Insert new leaves that represent the suffixes of $W_{i+1} = T[i+1..i+d]$ that appear exactly once in W_{i+1}, and
(3) After the leaf deletion due to Task (1) and each leaf insertion due to Task (2), maintain the label $\langle \ell, r \rangle$ of every edge as a *valid* pair in the new window $W_{i+1} = T[i+1..i+d]$, such that $i+1 \leq \ell \leq r \leq i+d$.

We do not scan all edges after each shift. Rather, whenever a local topological update creates, modifies, or needs to materialize an edge label, we compute a fresh index-pair for that edge in $O(1)$ worst-case time. While Task (2) also appears in online suffix tree constructions [10], Tasks (1) and (3) are specific to the sliding window model.

Larsson [5,6] pointed out that Task (1) is a mere operation that takes $O(1)$ *worst-case* time. Larsson [5,6] also showed that Task (2) can be done in $O(1)$ *amortized* time per character, by adapting Ukkonen's online suffix tree construction [10].

The focus of this paper is Task (3) of updating edge labels, for which the following two existing approaches are known:

- The first one is the so-called *credit-based* approach proposed by Larsson [5,6], that is based on the ideas from Fiala and Greene's earlier work [3]. The basic idea of this method is to propagate a credit to each edge label that needs to be updated, and it can be shown that the time cost for updating each edge label $\langle \ell, r \rangle$ takes $O(1)$ *amortized* time. The worst-case time is however $O(d)$ per left-end character deletion and right-end character insertion. In addition, in Section 5 we show that the worst-case time of the credit-based method is $\Omega(d)$ for a single leaf insertion or deletion on some string.
- The second one is the so-called *batch-based* approach proposed by Senft [9]. Senft's method uses standard batch-updates on overlapping blocks of size $2d$ over the text T, shifting a window of size d in each block to the right, reconstructing the suffix tree from scratch when the window no longer fits in the block, moving on to the next block prefixed by the window. It is clear that the batch-based approach takes $O(1)$ *amortized* time. However, the worst-case time is $O(d)$ per character deletion/insertion as well as for leaf deletion/insertion.

In this paper, we present the first method, named the *leaf-pointer* based approach, that is able to perform Task (3) in $O(1)$ *worst-case* time for each edge label update that is incidental to a leaf insertion or deletion. We first make a simple observation that maintaining the label for each edge $u \to v$ reduces to maintaining a pointer from node v to a leaf in the subtree rooted at v (Lemma 2 in Section 3). Then, we present how to maintain a pointer from each node to a specific leaf in its subtree in $O(1)$ worst-case time (Section 4). This is achieved by introducing our concepts of *primary* and *secondary* nodes and by maintaining leaf pointers built on them, which may be of independent interest and may be used in other applications. Table 1 summarizes the comparison of the existing methods and our proposed method.

Related work. Brodnik and Jekovec [1] used a similar notion to our leaf pointers. They showed how to perform pattern matching in $O(m \log \sigma + occ)$ time on the sliding suffix tree with $O(d)$ working space, where m is the query pattern length and occ is the number of *all* pattern occurrences in the current window. Since Brodnik and Jekovec [1] utilize the credit-based algorithm to maintain their leaf pointers, their method takes $O(1)$ *amortized* time for each leaf insertion/deletion.

Table 1. Comparison of algorithms for maintaining a suffix tree of a sliding window of size d on a string of length n over an alphabet of size σ. In the table, "amortized" and "worst case" refer to "amortized $O(1)$ time" and "worst case $O(1)$ time", respectively. The $O(\log \sigma)$ term in the total time complexities is due to maintaining branches of nodes.

algorithms	Task (1)	Task (2)	Task (3) per node deletion/insertion	total time	working space
credits [5,6]	worst case	amortized	amortized	$O(n \log \sigma)$	$O(d)$
batch [9]	worst case	amortized	amortized	$O(n \log \sigma)$	$O(d)$
leaf pointers [ours]	worst case	amortized	worst case	$O(n \log \sigma)$	$O(d)$

2 Preliminaries

2.1 Strings

Let Σ be an alphabet of size σ. An element of the set Σ^* is a string. The length of a string w is denoted by $|w|$. For a string $w = pts$, p, t, and s are called a *prefix*, *substring*, and *suffix* of w, respectively. A suffix s of w that occurs twice or more in w is said to be *repeating* in w. For a string w, $w[i]$ denotes the i-th character of string w for $1 \leq i \leq |w|$, and $w[i..j]$ denotes the substring $w[i]w[i+1] \cdots w[j]$ of string w for $1 \leq i \leq j \leq |w|$.

2.2 Sliding suffix trees

The suffix tree [11,4] of a string T, denoted $\mathsf{STree}(T)$, is the compacted trie of all suffixes of T. Note that each non-leaf node of $\mathsf{STree}(T)$ has at least two children (unless T is a unary string), and thus, the number of nodes in $\mathsf{STree}(T)$ is linear in the length of T. Each edge is labeled by a non-empty substring of T and the labels of all outgoing edges of the same node begin with distinct characters. Any substring w of T can be associated with a corresponding location (either on a node or a position on an edge) in the suffix tree where the concatenated path label from the root to the location is w. Depending on the context, we will use strings and their locations in the suffix tree interchangeably. An edge of a suffix tree from node u to node v is denoted by $u \to v$. The string label x for each edge $u \to v$ is represented by an index-pair $\langle \ell, r \rangle$ such that $x = T[\ell..r]$. This way, $\mathsf{STree}(T)$ can be stored in space linear in the length of T. The sliding suffix tree [3,9,1] is the suffix tree for a sliding window $W_i = T[i..i+d-1]$ of size d, that shifts over the input string T of length n from left to right, for increasing $i = 1, \ldots, n-d+1$. This is a version of suffix trees a.k.a. Ukkonen trees [10], in which non-repeating suffixes of W_i are represented by leaves while repeating suffixes of W_i are represented by (possibly implicit, non-branching) internal nodes. Let us consider a sliding window $W_i = T[i..i+d-1]$ over an input string T. For ease of description, let $\ell_i = i$ and $r_i = i+d-1$. Let $W_i[s..r_i]$ be the longest repeating suffix of W_i. For any $\ell_i \leq k < s$, $\mathit{leaf}_i(k)$ denotes the leaf of the suffix tree of W_i that represents the suffix $W_i[k..r_i]$. The sliding suffix tree $\mathsf{STree}(W_i)$ for W_i consists of the leaves $\mathit{leaf}_i(k)$ for $\ell_i \leq k < s$, branching internal nodes, and the root. When there is no risk of confusion, we simply denote it by $\mathit{leaf}(k)$. Figure 1 shows an example of how a sliding suffix tree changes between two iterations. On the left side of Figure 1, strings $\mathtt{a}, \mathtt{abaca}$ and the empty string are considered to be located on nodes, while strings $\mathtt{bac}, \mathtt{c}$ are considered to be located on edges.

In our sliding suffix tree algorithm, only the last d characters of T are explicitly stored to achieve $O(d)$ space efficiency, as done in the credit-based approach by Larsson [5,6]. Thus, any edge whose index-pair contains an index $k < \ell_i$, or in other words

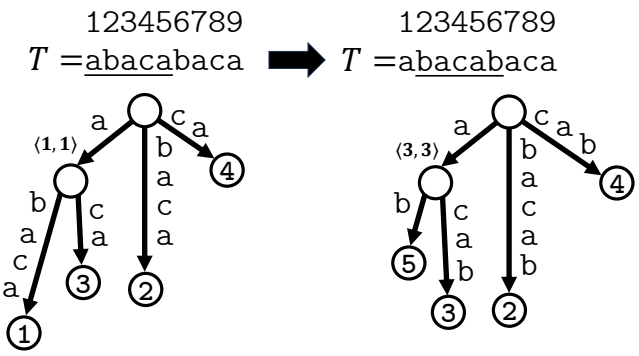

Figure 1. Sliding suffix tree across two iterations with $T =$ `abacabaca` and $|W| = 5$. *leaf*(1) is deleted in the second iteration due to its corresponding suffix `abaca`, which would grow into `abacab`, not existing in the new window. The edge labeled `a` shows an example of an edge whose index-pair becomes outdated; if it was represented by $\langle 1, 1\rangle$ in the first iteration, an update is required as the index 1 contained in the interval is no longer inside the window.

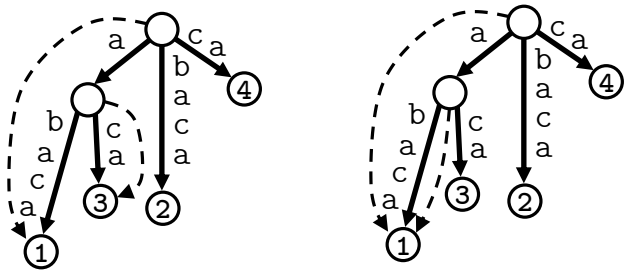

Figure 2. Two possible configurations of leaf pointers for the suffix tree of `abaca`. The dashed arrows depict leaf pointers.

any edge with an index-pair corresponding to an occurrence of the edge-label that is no longer entirely inside the window, needs to be updated to a *fresh* index-pair $\langle \ell, r\rangle$ such that $[\ell..r] \subseteq [\ell_i..r_i]$. A *leaf pointer* on a suffix tree is defined as a pointer from each internal node to any of its descendant leaves. Leaf pointers were originally proposed by Brodnik and Jekovec [1] for their use in online pattern matching. Figure 2 shows configuration examples of leaf pointers. Unlike with online suffix trees, maintaining them for a sliding suffix tree is not a trivial task. Brodnik and Jekovec [1] claimed that one can update each leaf pointer in $O(1)$ amortized time per iteration, by extending the credit-based method by Larsson [5,6].

In the rest of this paper, we will prove the following theorem:

Theorem 1. *Let T be a string of length n and d be the size of the sliding window $W_i = T[\ell_i..r_i]$ over T. There is an algorithm of $O(d)$ space that maintains a leaf pointer of every node in the sliding suffix tree in $O(1)$ worst-case time per update.*

3 Getting fresh index-pairs from leaf pointers

We assume that the topology of the suffix tree is maintained, along with the string length corresponding to each node and the index k for each *leaf*(k). Provided that leaf pointers are maintained, we can find a fresh index-pair of any edge $u \to v$ with edge label x in constant time as follows: If v is a leaf, it is clear that $\langle k_v, r_i\rangle$ is a fresh index-pair of the edge where $v =$ *leaf*(k_v) and r_i is the right-end of the current

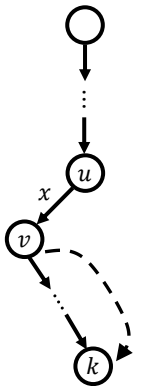

Figure 3. A figure showing the relationship between u,v, and $leaf(k)$ in the context of computing the index-pair in the general case.

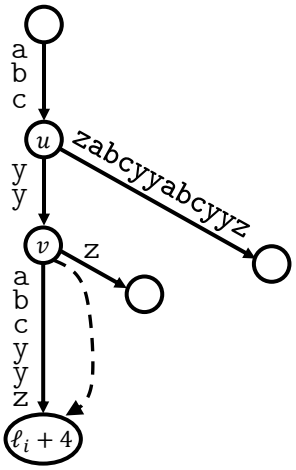

Figure 4. The part of the suffix tree that starts with the character a in Example 3.

window. Otherwise, traverse the leaf pointer of $v = ux$ to the destination leaf $leaf(k)$. The existence of the leaf implies that the suffix $T[k..r_i]$ must occur inside the current window W_i, and ux is a prefix of this suffix. Thus, $\langle k + |u|, k + |ux| - 1\rangle$ is a fresh index-pair of $u \to v$, corresponding to the substring x. Additionally, the edge label $\langle k + |u|, k + |ux| - 1\rangle$ is also *strongly fresh*, meaning that the preceding u at start-index k is also inside the window. Figure 3 illustrates the relationship between u,v, and $leaf(k)$. The following lemma immediately follows:

Lemma 2. *Assume that the leaf pointers for all internal nodes are maintained. Then, for any edge $u \to v$, one can obtain in $O(1)$ worst-case time an index-pair $\langle \ell, r\rangle$ such that $T[\ell..r]$ matches the edge's label, and $[\ell - |u|..r] \subseteq [\ell_i..r_i]$.*

Example 3. Consider $W_i =$ `abczabcyyabcyyz` and $T = PW_i$ for some $P \in \Sigma^*$, $u =$ `abc`, $v = ux =$ `abcyy`. Then, u and v are branching nodes. Regardless of the content of P, we can get an index-pair of the edge $u \to v$ with label $x =$ `yy` as follows: Traverse the leaf pointer of v to arrive at one of its descendant leaves, say, $leaf(\ell_i + 4)$ corresponding to the suffix `abcyyabcyyz`. Then, we get the index-pair $\langle \ell_i + 4 + |u|, \ell_i + 4 + |u| + |x| - 1\rangle = \langle \ell_i + 7, \ell_i + 8\rangle$, which gives us the correct label, since $W_i[8..9] =$ `yy`. Figure 4 shows the relevant part of the suffix tree of W_i, namely the locations starting with `a`.

We note that, even if the edge $u \to v$ with index-pair $\langle \ell, r\rangle$ is a fresh edge in the weak sense, i.e., $[\ell..r] \subseteq [\ell_i..r_i]$, we cannot use the opposite of the above method to obtain a leaf pointer of node u. We can obtain the start-index $\ell - |u|$, but the corresponding $leaf(\ell - |u|)$ may already be deleted, i.e., $\ell - |u| < \ell_i$, unless the edge label is strongly fresh, which cannot be assumed for sliding suffix tree construction algorithms in general.

Theorem 1 and Lemma 2 immediately lead to the following:

Corollary 4. *Let T be a string of length n and d be the size of the sliding window $W_i = T[\ell_i..r_i]$ over T. There is an algorithm of $O(d)$ space that resolves Task (3) in $O(1)$ worst-case time per leaf insertion/deletion.*

We will prove Theorem 1 in Section 4.

4 Maintaining leaf pointers in $O(1)$ worst-case time

Our leaf pointer algorithm maintains the following invariants: For each internal node x with $c > 0$ children, exactly one child node is designated as a *primary* node, while each of the remaining $c - 1$ children are designated as *secondary* nodes. An edge is similarly called primary or secondary depending on its destination node. For convenience, the root will be considered a secondary node.

Let S be the set of (arbitrarily chosen) primary nodes. Then, for each secondary internal node x, we maintain the *primary leaf pointer* $PLP_S(x)$ that points to the leaf node reached by traversing down the primary edges from x. It is clear that the path of primary edges from a secondary node x leads to a leaf (see also Figure 5 for illustrations). For each secondary leaf node x, we let $PLP_S(x) = x$. When the choice S of primary nodes in the sliding suffix tree is clear from the context, we abbreviate $PLP_S(x)$ as $PLP(x)$.

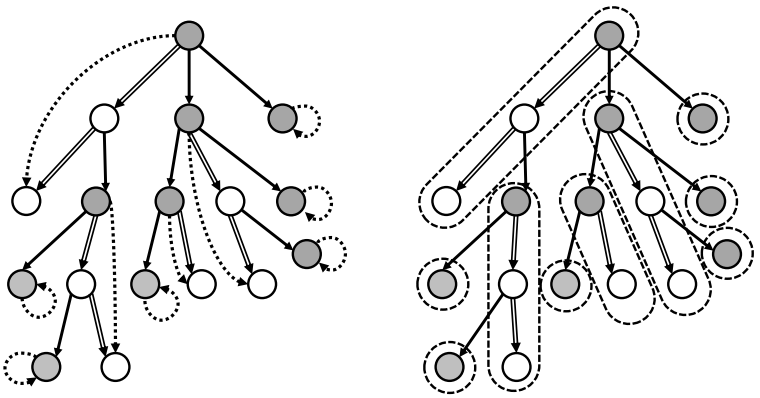

Figure 5. The white and gray nodes represent primary and secondary nodes, respectively. The double-lined solid arrows represent primary edges, while the single-lined solid arrows represent secondary edges. Left: The dotted arrow from each secondary node x represents the primary leaf pointer $PLP(x)$. Right: Our primary/secondary edges induce a path-decomposition of the tree which can be seen as a relaxed version of the well-known heavy-path decomposition, where our primary edge of each node can be *any* of its outgoing edges.

Assuming the above invariants are maintained, we can answer the leaf pointer query for any internal node x as follows: If x is secondary, return $PLP(x)$. Otherwise, x is a primary (thus non-root) internal node, meaning it has at least two children, so it must have a secondary child y. In this case, return $PLP(y)$ for an arbitrarily chosen secondary child y. See Figure 6 for a concrete example.

The invariants above also guarantee that each leaf only has one primary leaf pointer pointing to it, making updates in constant time and linear space feasible.

Lemma 5. *For each leaf u of the tree, there is exactly one primary leaf pointer that points to u. Equivalently, the primary paths induce a one-to-one correspondence between secondary nodes and leaves. Hence every leaf has a unique inverse PLP pointer.*

Proof. By categorizing the edges of the tree into primary or secondary ones, one can induce a decomposition of the tree to a disjoint set of paths (see also Figure 5). Here, each path $\mathcal{P}$ in the decomposition begins with a secondary node x, continues with a series of primary nodes (possibly empty), and ends with a leaf u. Thus, the leaf u is pointed by the primary leaf pointer from the secondary node x that is the beginning

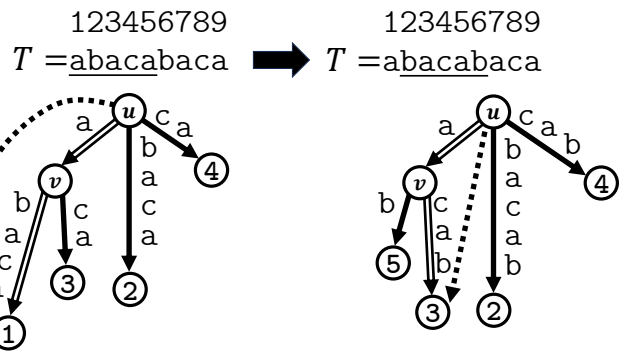

Figure 6. Primary leaf pointers maintained across two iterations of a sliding suffix tree. Primary edges are depicted by double-lined arrows, secondary edges by single-lined arrows, and primary leaf pointers by dotted arrows. The primary leaf pointers from each secondary leaf that point to themselves are omitted in this figure. Left (before the update): The secondary branching node u returns $PLP(u) = leaf(1)$, while a leaf pointer query on the primary branching node v returns the primary leaf pointer of its secondary child $leaf(3)$, resulting in $PLP(leaf(3)) = leaf(3)$. Right (after the update): Queries on u and v return $leaf(3)$ and $leaf(5)$, respectively.

node of the path $\mathcal{P}$. It is clear that the leaf u is pointed only by this secondary node x that belongs to the same path $\mathcal{P}$. □

When the suffix tree for $W_{i-1} = T[\ell_{i-1}..r_{i-1}]$ is updated to the suffix tree for $W_i = T[\ell_i..r_i]$ at the i-th iteration, the structure of the suffix tree can change. This structural change requires us to maintain primary leaf pointers efficiently, as some leaves and internal nodes can be removed and/or added. The following property is known:

Lemma 6 (Larsson [5]). *For any string T of length n, the total number of nodes removed and/or added is $O(n)$ regardless of the (fixed) size of the sliding window.*

Below, we describe our algorithm which maintains primary leaf pointers in $O(1)$ worst-case time per leaf inserted into or deleted from the sliding suffix tree. This together with Lemma 6 will give us an $O(n)$ total time updates of the sliding suffix tree.

The key to our efficient algorithm is Lemma 5 which states that the primary leaf pointer function $PLP(\cdot)$ is an injection. Namely, only a constant number of primary leaf pointers need to be updated per leaf insertion/deletion. We will also assume that the inverse pointers of PLPs are maintained, so that when $PLP(x) = u$ we can also access x from u in constant time. Whenever we create, delete, or redirect a PLP, we also update its inverse pointer accordingly.

In what follows, the *primary path* of a PLP from a secondary node x denotes the path consisting of the primary edge(s) from x to the leaf destination of $PLP(x)$. Similarly, the primary path of any node x refers to the path of primary edges from x to a leaf.

4.1 Handling node insertions on nodes

Our procedure for inserting leaf u as a child of an existing internal node w is as follows:

Case 1-1: When w has no child (Figure 7).

Procedure: Make u primary and set $PLP(w) = u$.
Correctness: This case only happens when w is the root because inserting a new leaf under an existing leaf does not occur in suffix trees of non-empty strings. The correctness is trivial.

Case 1-2: When w has a child (Figure 8).
Procedure: Make u secondary and initialize $PLP(u) = u$.
Correctness: In this case, no existing primary path changes, and therefore no existing PLP needs to be updated. A new secondary node u is created, thus $PLP(u)$ needs to be created. Since u is a leaf, u itself is the correct destination of $PLP(u)$.

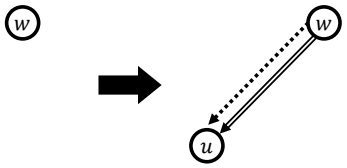

Figure 7. Case 1-1.

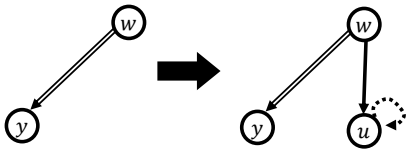

Figure 8. Case 1-2.

4.2 Handling node insertions on edges

Here, we describe our procedure for inserting a leaf u, such that an edge $x \to y$ is split into $x \to w$ and $w \to y$, where w is a newly created branch node and is the parent of u.

Case 2-1: When y is primary (Figure 9).
Procedure: Make w primary, u secondary, and initialize $PLP(u) = u$.
Correctness: In this case, the primary path that includes $x \to y$ is split. Thus, the primary path that contained $x \to y$ has changed. However, its destination clearly remains the same, namely the leaf at the end of the primary path of y. Thus, no change for existing PLPs is needed. Here, again a secondary leaf node u is created, and thus its PLP needs to be created to point to itself.

Case 2-2: When y is secondary (Figure 10).
Procedure: Make w secondary, u primary, and initialize $PLP(w) = u$.
Correctness: In this case, the edge being split $x \to y$ is secondary, thus no existing PLP changed its primary path. Additionally, a new secondary node w is created, so $PLP(w)$ needs to be created and initialized. Since y is made to remain secondary, the primary child of w has to be the newly created u. Since u is a primary leaf child of w, it is the correct destination of $PLP(w)$.

4.3 Handling leaf deletions under root parent node

Here, we describe our procedure for deleting a leaf u, when its parent node w is the root. Note that no merging occurs in Cases 3-1–3-3 since w has no parent.

Case 3-1: When u is primary and w has no other children (Figure 11).
Procedure: Set $PLP(w) = w$.
Correctness: In this case, the tree consists only of the root after the update, thus the only PLP that exists is from the root w pointing to itself.

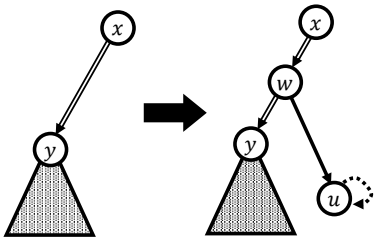

Figure 9. Case 2-1.

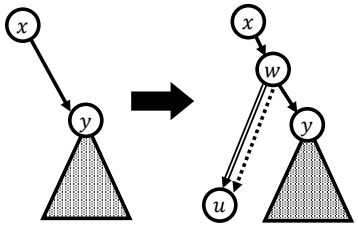

Figure 10. Case 2-2.

Case 3-2: When u is primary and w has at least one other child (Figure 12).
Procedure: Make y primary and set $PLP(w) = v$, where y is any secondary child of w and v is the leaf that $PLP(y)$ points to.
Correctness: Here, the primary child u of w is deleted, but w has at least one more child y remaining. Thus, y is changed into primary to maintain the invariant of child-having nodes having exactly one primary child. Consequently, y no longer has a PLP after the update. Also, the primary path from w changes into $w \to y \to \cdots$. Since the subtree of y remains unchanged, clearly the destination of this primary path, which is also the correct new destination of $PLP(w)$, is the former destination of $PLP(y)$ before the update.
Case 3-3: When u is secondary (Figure 13).
Procedure: No further updates.
Correctness: Since the node and edge deleted are secondary, and since no merging occurs, all existing primary paths remain the same, necessitating no change in existing PLPs that remain (i.e., undeleted).

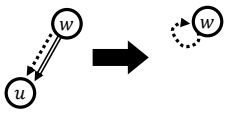

Figure 11. Case 3-1.

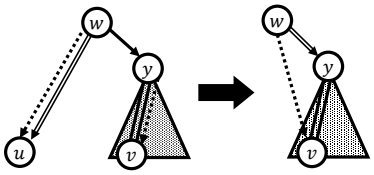

Figure 12. Case 3-2.

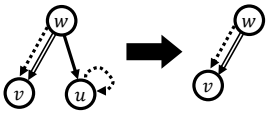

Figure 13. Case 3-3.

4.4 Handling primary leaf deletions under non-root parent node

Here, we describe our procedure for deleting a primary leaf u, when its parent node w is not the root. Let x be the parent of w, y any secondary child of w, v the leaf $PLP(y)$ points to, and z the node such that $PLP(z) = u$. Since w is not the root, it has at least two children.

Case 4-1: When w has more than two children (Figure 14).
Procedure: Make y primary, and set $PLP(z) = v$.
Correctness: In this case, z is w or its ancestor, and w does not get deleted. Here, the deletion of the primary node u and its incoming primary edge necessitates the appointment of another child of w to become primary, and also induces a change on the primary path of the PLP that pointed to u, namely that from z. Similarly to Case 3-2, y is chosen as the new primary child of w and thus the primary path from z now goes into y, and due to the unchanged subtree of y, v is the correct new destination of $PLP(z)$.

Case 4-2: When w has two children and is primary (Figure 15).
Procedure: Make y the primary child of x, and set $PLP(z) = v$.
Correctness: In this case, z is x or its ancestor. Since w, which was primary, becomes non-branching and gets deleted due to merging, its parent x needs to appoint a new primary child, for which y is chosen. Thus, the primary path from z now goes into y, and again we know the primary path destination from y is v, the former destination of $PLP(y)$ before the update, and therefore v is the correct destination of $PLP(z)$.
Case 4-3: When w has two children and is secondary (Figure 16).
Procedure: Make y a secondary child of x.
Correctness: In this case, $z = w$, and w gets deleted due to merging. Since w is secondary, the only affected primary path is the deleted path from w to u; all primary paths corresponding to remaining PLPs stay unchanged.

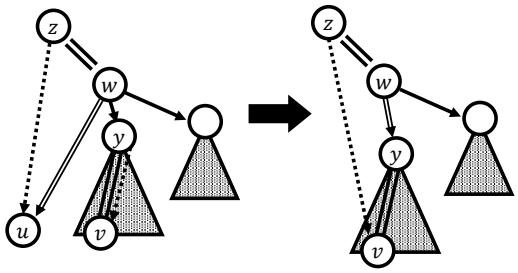

Figure 14. Case 4-1.

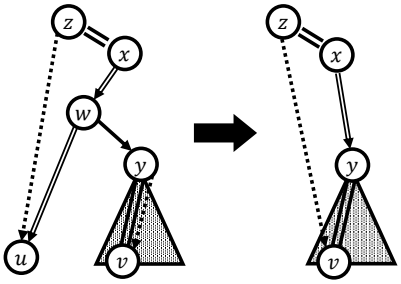

Figure 15. Case 4-2.

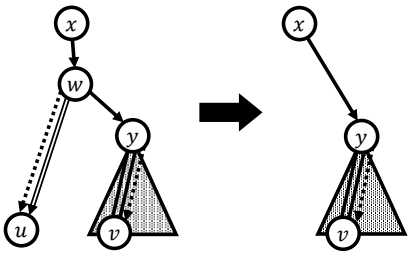

Figure 16. Case 4-3.

4.5 Handling secondary leaf deletions under non-root parent node

Here, we describe our procedure for deleting a secondary leaf u and its parent node w is not the root. Let x be the parent of w and y the primary child of w. Since w is not the root, it has at least two children.

Case 5-1: When w has more than two children (Figure 17).
Procedure: No further updates.
Correctness: In this case, w does not get deleted. Since only a secondary node is deleted and no merging occurs, no existing primary paths change.
Case 5-2: When w has two children and is primary (Figure 18).
Procedure: Make y the primary child of x.

Correctness: In this case, the deletion of u causes w to be deleted due to merging, and thus the primary path that included $x \to w \to y$ changes. However, the destination remains the same, namely the destination of the primary path from y. Thus, no existing PLPs change. Additionally, x remains having exactly one primary child, by letting y remain primary.

Case 5-3: When w has two children and is secondary (Figure 19).

Procedure: Make y a secondary child of x and set $PLP(y) = v$, where v is the leaf pointed to by $PLP(w)$.

Correctness: Here, the deletion of u causes w to be deleted due to merging. Since y is made secondary, the primary child of x remains unchanged, but $PLP(y)$ needs to be initialized. Since y was formerly primary, the former destination of $PLP(w)$ is in the unchanged subtree of y, namely leaf v, and therefore v is the correct destination of the new $PLP(y)$.

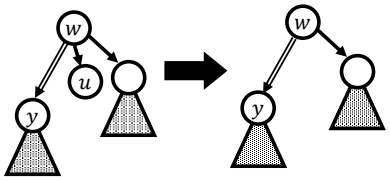

Figure 17. Case 5-1.

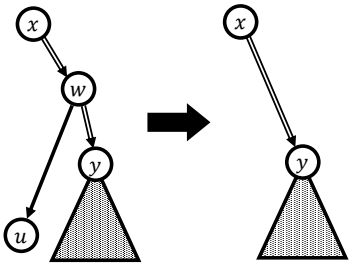

Figure 18. Case 5-2.

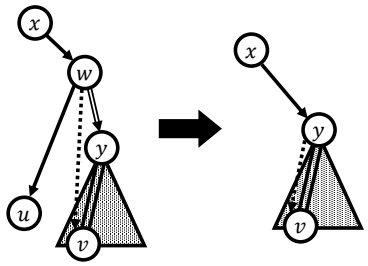

Figure 19. Case 5-3.

To illustrate the whole picture of our proposed method, we show in Figures 20 and 21 the decision trees of our leaf-pointer maintenance algorithm for the cases of right-end insertions and left-end deletions, respectively.

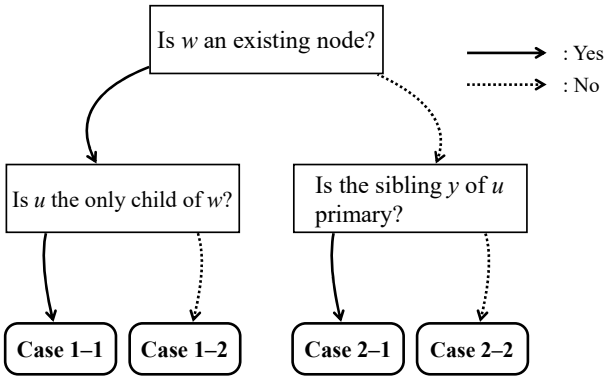

Figure 20. The decision tree representing the four cases of Sections 4.1 and 4.2, when a new node is inserted into the suffix tree where u is the new leaf and w is the parent of u.

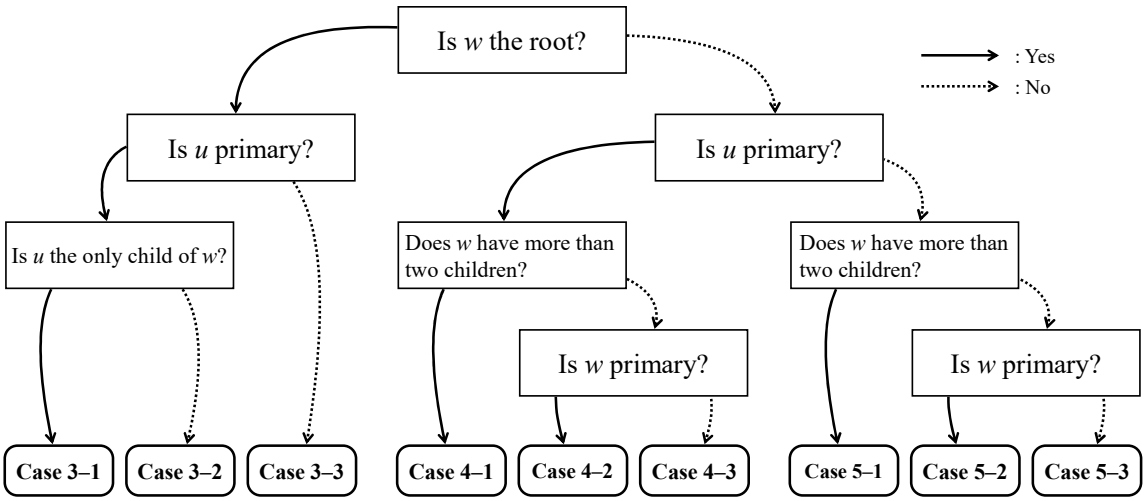

Figure 21. The decision tree representing the nine cases of Sections 4.3, 4.4, and 4.5, when an old leaf is deleted from the suffix tree where u is the leaf to be deleted and w is the parent of u.

We conclude this section with the following theorem:

Theorem 7. *Our algorithm described above correctly maintains leaf pointers on the sliding suffix tree for $W_i = T[\ell_i..r_i]$ in $O(1)$ time per node insertion/deletion, using $O(d)$ working space.*

Proof. The described steps to maintain the invariants cover all possible cases of insertion and deletion in a sliding suffix tree, and each step can clearly be handled in $O(1)$ worst-case time. This can also be observed from our decision trees in Figures 20 and 21. The correctness of all the cases has already been shown. This allows us to answer any leaf pointer query in $O(1)$ worst-case time. The working space is clearly $O(d)$. □

We remark that our leaf-pointer maintenance algorithm works in $O(1)$ worst-case time even when the window size d is not fixed, and the working space is linear in the current window size.

5 $\Theta(d)$-time worst-case lower bound for credit-based method

In this section, we present an instance that requires $\Theta(d)$ worst-case time per single leaf insertion or deletion with the credit-based method [5,6], where d is the window size.

Here, we briefly recall the credit-based method by Larsson [5,6], with a correction by Senft [9] for leaf deletions. When a leaf is created, then it issues a *credit* to its parent. Each internal node u maintains a credit counter $\mathit{credit}(u) \in \{0, 1\}$. We initially set $\mathit{credit}(u) = 0$ when an internal node u is created.

When an internal node u receives a credit from a child v, then there are two cases:

1. If $\mathit{credit}(u) = 0$, then we set $\mathit{credit}(u) = 1$.
2. Otherwise (if $\mathit{credit}(u) = 1$), then we first spend the existing credit in u by setting $\mathit{credit}(u) = 0$, and update the label of the edge leading to u. Second, we pass, to the parent of u, the credit that came from the child v.

When an internal node v is deleted and two adjacent edges $u \to v$ and $v \to w$ are merged into a single edge $u \to w$, then two credits are issued from v independently of the value of $\mathit{credit}(v)$. Then, one credit is used to update the label of the edge $u \to w$, and the other credit is passed to the parent u of the deleted node v.

While we omit the details in this paper, each edge label update can be done in constant time, and the above procedure keeps all edge labels valid (see [5,6,9] for details). At most n leaves are created and deleted for all sliding windows W_i with $1 \leq i \leq n-d+1$ over a string of length n. Since each leaf issues at most three credits (one for its insertion and two for its deletion), the total number of credits issued is at most $3n$. Since each edge label update spends a credit, the time for updating an edge label is amortized $O(1)$ per leaf insertion or deletion.

In the next theorem, we show that this credit-based method requires $\Theta(d)$ time per leaf insertion or deletion in the worst case.

Theorem 8. *There exists a string for which Larsson's credit-based sliding suffix tree algorithm [5,6] requires $\Theta(d)$ edge-label updates for some single leaf insertion or deletion, where d is the window size.*

Proof. The following credit configuration is reachable by the standard Ukkonen-style insertion phase used in the sliding suffix tree algorithm. Here we measure the worst-case cost of an individual leaf-insertion suboperation occurring during one window shift; the deletion suboperation in the same shift is not charged to this leaf insertion.

For the case of leaf insertion, consider a string that has $W = a^k b a^{k-1}$ as its prefix of length $2k = d$. We then append a new character c to the right end of W. See also Figure 22 for illustrations. Since our string begins with $W = a^k b a^{k-1}$, we first build $\mathsf{STree}(W)$ in an online manner, from left to right, as is done by Ukkonen's construction [10]. Below, we explain how the sliding suffix tree is built and updated.

- Build $\mathsf{STree}(a^k)$ that consists only of a single path (see also the first step of Figure 22).
- Then, we update it to $\mathsf{STree}(a^k b)$, in which the new leaves $a^h b$ and their parents a^h are inserted in descending order for $h = k-1, \ldots, 1$ (see also the second step of Figure 22). When the internal node a^h is inserted, we initially set $\mathit{credit}(a^h) = 0$.

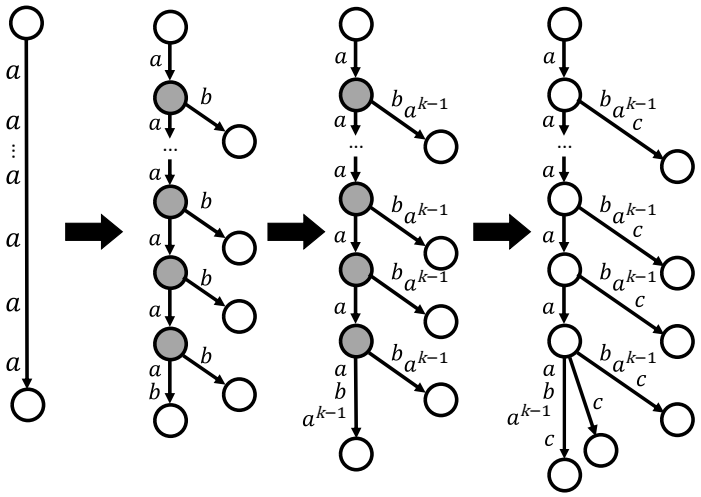

Figure 22. Illustration of $\Theta(d)$-time update of credit-based methods per single leaf insertion. The gray and white nodes represent internal nodes with credit value 1 and 0, respectively. Note that the subtrees starting with characters other than a are omitted for clarity.

As soon as the leaf $a^h b$ is inserted as a new child of a^h, then a^h receives a credit from this new leaf and we set $\mathit{credit}(a^h) = 1$. Thus, every node a^h for $1 \leq h \leq k-1$ obtains $\mathit{credit}(a^h) = 1$ on $\mathsf{STree}(a^k b)$.

- Next, we update $\mathsf{STree}(a^k b)$ to $\mathsf{STree}(a^k b a^{k-1})$ (see also the third step of Figure 22). No new leaves are inserted during this update.
- Finally, we update $\mathsf{STree}(a^k b a^{k-1})$ to $\mathsf{STree}(a^k b a^{k-1} c)$ (see also the fourth step of Figure 22). The first new leaf inserted is $a^{k-1}c$, which is a new child of a^{k-1}. Since $\mathit{credit}(a^{k-1}) = 1$, we reset it as $\mathit{credit}(a^{k-1}) = 0$ and pass the credit from the child $a^{k-1}c$ to the parent a^{k-2}. This propagates to all a^h in decreasing order of $h = k-1, \ldots, 1$.

Thus, the credit-based method requires $\Theta(d)$ edge-label updates for a single leaf insertion in the worst case.

For the case of leaf deletions, we consider a string $W' = a^d b$ of length $d+1$. See also Figure 23 for illustrations. We have $\mathit{credit}(a^h) = 1$ for all internal nodes a^h with $1 \leq h \leq d-1$ in $\mathsf{STree}(a^d b)$. Then we delete the leftmost character a from W' and update $\mathsf{STree}(a^d b)$ to $\mathsf{STree}(a^{d-1} b)$. This deletes the leaf corresponding to the longest suffix $a^d b$ and its parent a^{d-1} that becomes non-branching. After the deletion of the internal node a^{d-1}, a credit is sent to its parent a^{d-2} such that $\mathit{credit}(a^{d-2}) = 1$. We reset $\mathit{credit}(a^{d-2}) = 0$ and pass the credit to its parent a^{d-3}. This propagates to all a^h for decreasing $h = d-1, \ldots, 1$. Thus, the credit-based method requires $\Theta(d)$ edge label updates for a single leaf deletion in the worst case. □

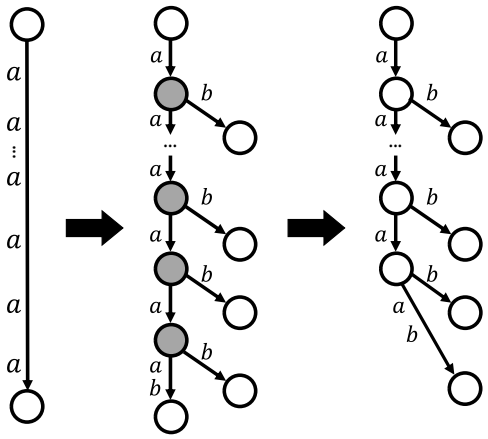

Figure 23. Illustration of $\Theta(d)$-time updates of credit-based methods per single leaf deletion.

Acknowledgements

The authors thank the anonymous referees of earlier versions of this paper for their helpful comments and suggestions for improvement. This work was supported by JSPS KAKENHI Grant Numbers JP23K24808, JP23K18466 (SI), JP24K02899 (HB), and JP24K20734 (TM).

References

1. A. Brodnik and M. Jekovec: *Sliding suffix tree.* Algorithms, 11(8) 2018, p. 118.
2. M. Crochemore, A. Héliou, G. Kucherov, L. Mouchard, S. P. Pissis, and Y. Ramusat: *Absent words in a sliding window with applications.* Inf. Comput., 270 2020, p. 104461.
3. E. R. Fiala and D. H. Greene: *Data compression with finite windows.* Commun. ACM, 32(4) 1989, pp. 490–505.
4. D. Gusfield: *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*, Cambridge University Press, Cambridge, 1997.
5. N. J. Larsson: *Extended application of suffix trees to data compression*, in Proceedings of the 6th Data Compression Conference (DCC '96), Snowbird, Utah, USA, March 31 - April 3, 1996, 1996, pp. 190–199.
6. N. J. Larsson: *Structures of String Matching and Data Compression*, PhD thesis, Lund University, Sweden, 1999.
7. T. Mieno, Y. Fujishige, Y. Nakashima, S. Inenaga, H. Bannai, and M. Takeda: *Computing minimal unique substrings for a sliding window.* Algorithmica, 84(3) 2022, pp. 670–693.
8. T. Mieno and S. Inenaga: *Space-efficient online computation of string net occurrences*, in 36th Annual Symposium on Combinatorial Pattern Matching, CPM 2025, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025, pp. 23:1–23:13.
9. M. Senft: *Suffix tree for a sliding window: An overview*, in 14th Annual Conference of Doctoral Students - WDS 2005, 2005, pp. 41–46.
10. E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.
11. P. Weiner: *Linear pattern matching algorithms*, in 14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973, 1973, pp. 1–11.
12. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Inf. Theory, 23(3) 1977, pp. 337–343.

Faster Indeterminate Pattern Matching Algorithm with Practical Implementations

Hamed Hasibi, Neerja Mhaskar*, and Tieyun Zhang

Algorithms Research Group,
Department of Computing & Software
McMaster University, Canada
hasibih@mcmaster.ca, pophlin@mcmaster.ca, nick.ty.zhang@gmail.com

Abstract. In computational biology, sequence analysis is complicated by genetic mutations, polymorphisms, and sequencing ambiguities, making exact string matching insufficient. Consequently, indeterminate pattern matching—where sequence positions may contain a set of possible characters—is a fundamental tool for identifying conserved biological motifs across degenerate sequences. In this paper, we present a simple and efficient *two-phase* solution to the indeterminate pattern matching problem, where both the text $\tilde{t}[1..n]$ and the pattern $\tilde{p}[1..m]$ contain indeterminate letters. Our approach leverages pattern matching algorithms for partial words, containing holes that match any character, yielding a solution that runs in $\mathcal{O}(n \log m + |Occ_{\diamond}| \cdot \min(m, D_1 + D_2))$-time, where $|Occ_{\diamond}|$ denotes the number of occurrences of $\tilde{p}$ in $\tilde{t}$ when all indeterminate letters are treated as holes, and D_1 and D_2 are the numbers of indeterminate letters in $\tilde{t}$ and $\tilde{p}$, respectively. This algorithm removes the exponential dependence on alphabet size in the best-known algorithm of Iliopoulos and Radoszewski [24] for indeterminate pattern matching over a constant-sized alphabet.
To bridge the gap between our theoretical bounds and practical hardware performance, we expand our *two-phase* theoretical approach into a highly efficient *three-stage* pipeline with the following stages: (I) heuristic candidate generation utilizing solid-letter anchors, (II) solid letter verification of the remaining solid positions, and (III) exact indeterminate letter resolution. We implement and compare three distinct heuristic wildcard matching strategies: a table-driven automaton (Modified SBNDM2), hardware-aware SIMD-accelerated scanning, and segment-based filtering. Through comprehensive experimental evaluation on DNA and protein datasets, we demonstrate that maximizing raw scanning throughput via the SIMD-accelerated approach fundamentally outperforms complex algorithmic filtering, yielding the best end-to-end performance. Finally, we verify the advantage of the heuristic implementations by comparing them with the FFT-based implementation, establishing that all three pipeline variants achieve a significant speedup over traditional convolution methods in biological scenarios.

Keywords: Indeterminate Pattern Matching, Partial Words, Bit-Parallelism, Boyer-Moore-Horspool, Single Instruction Multiple Data (SIMD), Simplified Backward Nondeterministic DAWG Matching (SBNDM), Fast Fourier Transform (FFT)

1 Introduction

Given a text $t[1..n]$ and a pattern $p[1..m]$, the pattern-matching problem is to identify all occurrences of p in t. This problem has been extensively studied for texts and patterns that are *solid strings*—strings containing exactly one symbol at each position—and several linear-time algorithms (with respect to the lengths of the strings) have been proposed [5,11,27,34]. The problem becomes significantly more challenging when

* Supported by Grant No. RGPIN-2024-06915 from the Natural Sciences & Engineering Research Council of Canada (NSERC).

Hamed Hasibi, Neerja Mhaskar, Tieyun Zhang: Faster Indeterminate Pattern Matching Algorithm with Practical Implementations, pp. 28–42.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

the input strings are *indeterminate or degenerate strings* [1,15,28], where a position may represent a set of multiple symbols rather than a single symbol.

The first pattern-matching algorithm for indeterminate strings was proposed by Abrahamson [1]. Since then, several algorithms have been developed for this problem [18,19,20,23,35]. In 2016, Iliopoulos and Radoszewski [24, Lemma 17] presented an $\mathcal{O}(n \log m)$-time algorithm based on the Fast Fourier Transform via convolution; this remains the fastest known running time for a constant-sized alphabet. Beyond standard exact matching, alternative paradigms for indeterminate strings have also been explored; for instance, Gawrychowski et al. [16,17] established NP-hardness for parameterized and order-preserving matching, while providing an expected $\mathcal{O}(n \log n \log \log n)$-time algorithm for Cartesian tree matching.

In recent years, the concept of *conservative degenerate* strings [2] has been introduced to model practical constraints in biological sequences. An indeterminate string is termed *conservative* if the number of indeterminate letters in the text and pattern is bounded by constants D_1 and D_2, respectively. Crochemore et al. [10] showed that this restriction yields a $\mathcal{O}((D_1 + D_2)n)$-time algorithm. Subsequently, Daykin et al. [12] proposed a $\mathcal{O}(n + D_1 m^2 + |Occ|)$-time algorithm for a constant-sized alphabet based on the Degenerate Burrows–Wheeler Transform.

In 1974, the study of pattern matching on strings with wildcards that match any symbol in the alphabet (a special case of indeterminate strings) began with the work of Fischer and Paterson [15], who proposed an $\mathcal{O}(n \log m \log |\Sigma|)$-time algorithm using the Fast Fourier Transform (FFT). A central challenge that emerged subsequently was eliminating the dependence on the alphabet size. In 1998, Indyk [25] introduced a randomized convolution-based algorithm that is independent of Σ and runs in $\mathcal{O}(n \log n)$ time. In 2002, Kalai [26] improved upon this result with a simpler randomized algorithm running in $\mathcal{O}(n \log m)$ time. That same year, Cole and Hariharan [9] achieved the first *deterministic* $\mathcal{O}(n \log m)$-time algorithm using convolution. Later, Clifford and Clifford [7] presented a simpler deterministic solution with the same time bound.

Numerous subsequent works have further expanded this area, addressing both theoretical improvements and practical considerations [3,4,6,8,32,30,31,33].

In this paper, we present a simple two-phase algorithm for pattern matching on indeterminate strings over a variable alphabet, running in $\mathcal{O}(n \log m + |Occ_\diamond| \min(m, D_1 + D_2))$ time. Phase I transforms the indeterminate text and pattern into partial words and applies the $\mathcal{O}(n \log m)$-time pattern matching algorithm of [7,9] to the transformed strings, generating candidate matches (which may include false matches). Phase II then eliminates these false matches in $\mathcal{O}(|Occ_\diamond| \min(m, D_1 + D_2))$ time.

Although FFT-based approaches provide strong theoretical guarantees, they are often less efficient in practice due to large constant factors and poor cache behaviour. Hence, we extend our *two-phase* approach into an efficient *three-stage* pipeline. In Stage I, candidate matches are generated heuristically using solid-letter anchors. Stage II verifies the remaining solid positions, and Stage III outputs the exact occurrences of the indeterminate pattern in the indeterminate text.

For Stage I, we adopt three heuristic wildcard matching strategies: a table-driven automaton (Modified SBNDM2), a hardware-aware SIMD-accelerated scanning method, and a segment-based filtering approach. We implement all three strategies and conduct extensive experiments on DNA and protein datasets. The results show that maximizing raw scanning throughput via SIMD acceleration consistently outperforms more complex filtering techniques, yielding the best overall performance. For completeness, we also implement our FFT-based solution for comparison. The results

confirm that the FFT-based approach is significantly slower in practice than all three *three-stage* pipeline implementations.

The paper is organized as follows. In Section 2, we introduce the necessary terminology. In Section 3, we present our two-phase indeterminate pattern matching solution. Section 4 presents the three-stage pipeline approach and its variants, while Section 5 provides experimental results, including performance analysis and discussion. Finally, Section 6 concludes the paper with a few open problems.

2 Preliminaries

Let Σ be a totally ordered set of *symbols* (or *letters*), called the *alphabet* of size σ. A *solid string* $w[1..n]$ is a sequence of n letters drawn from Σ. The length of a string $w[1..n]$ is denoted by $|w| = n$. Given a string $x = uvw$, u is called a *prefix* of x, v is called a *substring* of x, and w is called a *suffix* of x.

Let Σ' be the set of all non-empty subsets of Σ. An element $\lambda \in \Sigma'$ is referred to as an *indeterminate* letter (or *non-solid* letter), if $|\lambda| \geq 2$; otherwise, it is referred to as a *solid letter*. An *indeterminate string* $\tilde{x}[1..n]$ (or *degenerate* string) is a string containing at least one indeterminate letter. A position i in $\tilde{x}$, for $1 \leq i \leq n$, is called *indeterminate* if $|\tilde{x}[i]| > 1$; otherwise, it is called *solid*. Two letters $\lambda_1, \lambda_2 \in \Sigma'$ are said to *match*, written as $\lambda_1 \approx \lambda_2$, if and only if $\lambda_1 \cap \lambda_2 \neq \emptyset$. Otherwise, they are said to *mismatch*, written as $\lambda_1 \not\approx \lambda_2$. An indeterminate string is called a *partial word* $x_\diamondsuit$ if each indeterminate letter in $x_\diamondsuit$ equals the entire alphabet Σ. In the context of partial words, these indeterminate letters are referred to as *wildcards*, *don't-cares*, or *holes*, and are denoted by $\diamondsuit$. Clearly, a wildcard symbol matches every letter in the string. The wildcard density *(dens)* of a partial word $x_\diamondsuit$ of length $|x|$ containing D wildcards is the ratio $D/|x|$. Given a text $t[1..n]$ and a pattern $p[1..m]$, a *window* at position i refers to the substring of the text of length m starting at index i, denoted as $t[i..i+m-1]$, where $1 \leq i \leq n-m+1$.

Throughout the paper, we represent solid strings by either lowercase letters x, y, z; indeterminate strings by placing a tilde over the strings $\tilde{x}, \tilde{y}, \tilde{z}$; and partial words by using the symbol $\diamondsuit$ as a subscript to x, y, z.

3 Two-phase indeterminate matching

In this section, we propose a two-phase algorithm for pattern matching on indeterminate strings. In Phase I, we convert the indeterminate text $\tilde{t}$ and pattern $\tilde{p}$ into partial words $t_\diamondsuit$ and $p_\diamondsuit$ by replacing each indeterminate letter with a wildcard ($\diamondsuit$). We then apply existing pattern-matching algorithms for partial words. Since wildcards may introduce false matches, Phase II filters these and retains only valid occurrences.

Lemma 1. *Let Occ be the set of all starting indices where $\tilde{p}$ occurs in $\tilde{t}$, $Occ_\diamondsuit$ the set of starting indices where $p_\diamondsuit$ appears in $t_\diamondsuit$, and Occ_f the set of all false occurrences. Then $Occ = Occ_\diamondsuit \setminus Occ_f$.*

Proof. We show $Occ \subseteq Occ_\diamondsuit$ by contradiction. Suppose there exists $e \in Occ$ but $e \notin Occ_\diamondsuit$. Since $e \in Occ$, we have $\tilde{p}[j] \approx \tilde{t}[e+j-1]$ for every position j in the window. Since $e \notin Occ_\diamondsuit$, there exists a position j such that $p_\diamondsuit[j] \not\approx t_\diamondsuit[e+j-1]$. This mismatch forces both $p_\diamondsuit[j]$ and $t_\diamondsuit[e+j-1]$ to be solid and unequal. By construction, a position is solid in $p_\diamondsuit$ (resp. $t_\diamondsuit$) if and only if it is solid in $\tilde{p}$ (resp. $\tilde{t}$), and in that

case the letter is copied unchanged; hence $\tilde{p}[j] = p_\diamondsuit[j]$ and $\tilde{t}[e+j-1] = t_\diamondsuit[e+j-1]$ are both solid singletons with $\tilde{p}[j] \neq \tilde{t}[e+j-1]$. Therefore $\tilde{p}[j] \cap \tilde{t}[e+j-1] = \emptyset$; that is, $\tilde{p}[j] \not\approx \tilde{t}[e+j-1]$, contradicting $e \in Occ$.

Finally, Occ_f is by definition the set of all false occurrences; that is, $Occ_f = Occ_\diamondsuit \setminus Occ$. Combined with $Occ \subseteq Occ_\diamondsuit$, this yields $Occ = Occ_\diamondsuit \setminus Occ_f$. □

Lemma 2. *([7,9]) Let $Occ_\diamondsuit$ be the set of all occurrences of the pattern $p_\diamondsuit[1..m]$ in the text $t_\diamondsuit[1..n]$. Then $Occ_\diamondsuit$ can be computed in $\mathcal{O}(n \log m)$ time.*

We apply Lemma 2 as a black box to the partial words $t_\diamondsuit$ and $p_\diamondsuit$ obtained in Phase I. For completeness, we briefly recall the approach underlying the lemma, due to Clifford and Clifford [7]. Each solid letter is mapped to a distinct positive integer and each wildcard $\diamondsuit$ to 0. For every alignment i, it evaluates the mismatch score

$$\sum_{j=1}^{m} p_\diamondsuit[j]\, t_\diamondsuit[i+j-1]\big(p_\diamondsuit[j] - t_\diamondsuit[i+j-1]\big)^2,$$

which is 0 if and only if $p_\diamondsuit[1..m] \approx t_\diamondsuit[i..i+m-1]$. Indeed, whenever either position contains a wildcard, the corresponding term vanishes, while for two solid letters the term is zero precisely when the letters are equal. Using the Fast Fourier Transform, the values of the above sum for all alignments $i \in [1..n-m+1]$ can be computed in $\mathcal{O}(n \log n)$ time. Partitioning the text into $\mathcal{O}(n/m)$ overlapping blocks of length $2m$ and applying the FFT-based procedure to each block yields the entire set $Occ_\diamondsuit$ in $\mathcal{O}(n \log m)$ time.

3.1 False Matching Elimination

Given $Occ_\diamondsuit$ from Phase I, Algorithm 1 computes Occ–the set of occurrences of $\tilde{p}$ in $\tilde{t}$. For each candidate position $e \in Occ_\diamondsuit$, Algorithm 1 verifies its validity by checking matches only at indeterminate positions in both the text and the pattern. To achieve this, it uses two auxiliary structures: an integer list p_{indet}, containing all indeterminate positions in $\tilde{p}$ and an integer array t_{next}, which stores, for each index position i, the *next* indeterminate position, if such a position exists; otherwise, it stores ∞:

$$t_{\text{next}}[i] = \begin{cases} \min\{\, j > i \mid |t[j]| > 1 \,\} & \text{if such a } j \text{ exists,} \\ \infty & \text{otherwise.} \end{cases} \tag{1}$$

Both structures can be computed by a simple linear scan of $\tilde{p}$ and $\tilde{t}$, respectively. For each index $e \in Occ_\diamondsuit$, the algorithm first rechecks matches for all indeterminate letters in $\tilde{t}[e..e+m-1]$, and then for all indeterminate letters in $\tilde{p}[1..m]$. The index e is added to Occ only if all checks succeed.

We now make explicit three key properties that establish the correctness of Algorithm 1. Fix a candidate $e \in Occ_\diamondsuit$.

(i) *The while loop (lines 11–14) checks exactly the indeterminate text positions in the window $\tilde{t}[e..e+m-1]$.* The index c_{index} is initialized to e if $\tilde{t}[e]$ is indeterminate and to $t_{next}[e]$ otherwise (lines 7–10), and is thereafter advanced by $c_{index} \leftarrow t_{next}[c_{index}]$. By Equation (1), t_{next} maps each index to the next indeterminate text position; hence the loop visits precisely the indeterminate positions of $\tilde{t}$ that lie in $[e, \min(n, e+m-1)]$, and no solid text position is ever examined.

Algorithm 1 Phase II Algorithm

```
1: Input: t̃[1..n], p̃[1..m], Occ_◇
2: Output: Occ                                   ▷ Occ is the list of occurrences of p̃ in t̃
3: Scan t̃ to compute t_next                      ▷ using Equation 1
4: Scan p̃ to compute p_indet                     ▷ p_indet is the ordered set of indeterminate positions in p̃
5: for all e ∈ Occ_◇ do
6:     match ← true
7:     if |t̃[e]| > 1 then
8:         c_index ← e                           ▷ c_index is the current index in t̃[e..e + m − 1]
9:     else
10:        c_index ← t_next[e]
11:    while c_index ≤ min(n, e + m − 1) do
12:        if (t̃[c_index] ≉ p̃[c_index − e + 1]) then
13:            match ← false; break
14:        c_index ← t_next[c_index]
15:    for all i ∈ p_indet do
16:        if (p̃[i] ≉ t̃[e + i − 1]) then
17:            match ← false; break
18:    if match then
19:        Occ.add(e)
20: return Occ
```

(ii) *The set p_{indet} covers all indeterminate pattern positions.* By construction (line 4), p_{indet} is the ordered set of all positions i with $|\tilde{p}[i]| > 1$, so the for loop (lines 15–17) checks every indeterminate position of $\tilde{p}$ and only those.

(iii) *e is accepted if and only if every position where at least one side is indeterminate matches.* By (i) and (ii), the union of positions inspected by the two loops is exactly the set of window positions at which $\tilde{t}$ or $\tilde{p}$ (or both) is indeterminate. Every remaining position is solid on both sides; since $e \in Occ_\diamond$, all such solid–solid positions already match. Therefore *match* stays *true* (and e is added to Occ) if and only if $\tilde{t}[e + j - 1] \approx \tilde{p}[j]$ at every indeterminate position; that is, if and only if $\tilde{p}$ occurs at e, so $e \in Occ$.

Together with Lemma 1, the key property (iii) shows that Algorithm 1 outputs exactly $Occ = Occ_\diamond \setminus Occ_f$, thereby establishing the correctness of our two-phase solution.

Scanning all elements of $Occ_\diamond$ requires $\mathcal{O}(|Occ_\diamond|)$ time. For each $e \in Occ_\diamond$, scanning the indeterminate positions in the substring $\tilde{t}[e..e + m - 1]$ and in the pattern $\tilde{p}$ requires at most $\mathcal{O}(\min(m, D_1 + D_2))$ time. Therefore, the total running time of Algorithm 1 is $\mathcal{O}(|Occ_\diamond| \cdot \min(m, D_1 + D_2))$. Thus, we get:

Lemma 3. *Phase II algorithm computes set Occ in $\mathcal{O}(|Occ_\diamond| \cdot \min(m, D_1 + D_2))$-time.*

By Lemmas 2 and 3, we get the following result.

Theorem 4. *Over a variable alphabet, the two-phase algorithm finds all occurrences of pattern $\tilde{p}[1..m]$ in text $\tilde{t}[1..n]$ in $\mathcal{O}(n \log m + |Occ_\diamond| \cdot \min(m, D_1 + D_2))$ time.*

It is worth noting that the complexity of our two-phase solution is independent of the alphabet size. In contrast, the algorithm by Iliopoulos and Radoszewski [24] with the best known running time includes a factor of 2^σ. Hence, for a variable-sized alphabet, it requires $\mathcal{O}(2^\sigma n \log m)$ time. The effectiveness of this algorithm deteriorates significantly as the alphabet size increases. Our approach eliminates this exponential dependence on σ, making it faster even for $\sigma \geq 4$.

4 Practical Three-Stage Pipeline and Variants

The two-phase solution of Section 3 is the theoretical foundation of our practical Three-Stage Pipeline approach. Recall that in Phase I, the FFT-based solution produces the set $Occ_\diamond$ in $\mathcal{O}(n \log m)$ time. However, the FFT based techniques incur large constant factor overheads for long patterns ($m \geq 256$), while modern hardware offers faster alternatives. Therefore, to improve execution time, we *decompose* Phase I into two stages: a fast heuristic candidate generator (Stage I) and a solid-letter verification filter (Stage II), while retaining Phase II as Stage III–yielding a three-stage pipeline. The correctness argument of Lemma 1 is unchanged because Stage II recovers the exact set $Occ_\diamond$ before Stage III is applied. We now describe this three-stage pipeline in detail:

1. **Stage I: Heuristic Pre-Filtering (Candidate Generation):** The goal of this stage is to produce a set Occ_{cand} that is guaranteed to contain $Occ_\diamond$ but may additionally contain false matches, which are filtered out in Stage II. To achieve high throughput, this stage uses a heuristic: instead of comparing the entire pattern against each text window, it checks only a small, carefully chosen subset of solid positions of $p_\diamond$, called an *anchor*.
   For each starting position i in $t_\diamond$, the algorithm checks whether $t_\diamond[i+j-1] \approx p_\diamond[j]$ holds for every position j in the anchor. If so, i is added to Occ_{cand}. Since the anchor covers only a strict subset of all pattern positions, this check is necessary but not sufficient for a true partial-word match. Consequently, $Occ_\diamond \subseteq Occ_{cand}$, and the difference $Occ_{cand} \setminus Occ_\diamond$ consists of false matches eliminated in Stage II.
2. **Stage II: Solid Letter Verification:** For each candidate position $i \in Occ_{cand}$, Stage II verifies the solid positions of $p_\diamond$ that were not covered by the anchor used in Stage I. Specifically, for each position $j \notin anchor$, where $p_\diamond[j]$ is a solid letter, the algorithm checks whether $t_\diamond[i+j-1] \approx p_\diamond[j]$. Candidates that fail any such check are discarded as false matches; those that pass all checks are added to $Occ_\diamond$. Since Stage II verifies all solid positions of $p_\diamond$ outside the anchor and Stage I verified the positions in anchor, then, every solid position of $p_\diamond$ has now been checked. Therefore, the set $Occ_\diamond$ produced at the end of this stage is exactly the set of positions where $p_\diamond$ occurs in $t_\diamond$, as guaranteed by Lemma 1.
3. **Stage III: Indeterminate Letter Resolution:** Algorithm 1 is applied to $Occ_\diamond$ to resolve indeterminate letter matches. For each $i \in Occ_\diamond$, the algorithm verifies that indeterminate letters of $\tilde{p}$ match the corresponding letters of $\tilde{t}[i..i+m-1]$, and vice versa. Candidates that fail this verification are discarded; those that pass are added to Occ, the set of occurrences of $\tilde{p}$ in $\tilde{t}$. The correctness of Stage III follows from Lemma 1.

To identify the most effective heuristic configuration for this pipeline, we explore three distinct paradigms for Stage I. Each of the implementations below employs a specific heuristic and parameterization to achieve a trade-off between speed and the number of candidates:

Modified SBNDM2 (Table-Driven Automaton): SBNDM2 (Simplified Backward Nondeterministic DAWG Matching) is a bit-parallel string-matching algorithm that scans the text right-to-left at each alignment within the text window using a Non-deterministic Finite Automaton (NFA) packed into a single machine word. We use the SBNDM2 variant from the SMART [1] framework [14] and adapt it to handle

[1] https://github.com/smart-tool/smart

wildcard positions. Its appeal for Stage I is that the NFA evaluates every solid position of $p_\diamond$ simultaneously, so it produces zero false matches at the Stage II boundary.

Preprocessing. SBNDM2 builds a table $B : \Sigma \to \{0,1\}^w$, where $B[c]$ is a w-bit integer with bit j set to 1 whenever $p_\diamond[m-j]$ equals c. For a wildcard position $m-j$ in $p_\diamond$, bit j is set in $B[c]$ for *every* $c \in \Sigma$, encoding that any letter in the text matches that position. The NFA window covers all positions within the first $\min(m, w)$ symbols of $p_\diamond$. Every solid position in this window is encoded in B, so the NFA check is necessary and sufficient for partial-word matching within the window—yielding zero false matches passed to Stage II for patterns of length $m \leq w$.

Searching. The algorithm maintains an NFA state $D \in \{0,1\}^w$. Processing the text right-to-left within a window starting at position i, each text letter $t_\diamond[i]$ updates the state as

$$D \leftarrow (D \ll 1) \ \& \ B[t_\diamond[i]].$$

A match at position i is reported when the most significant bit of D is set. Because D encodes a set of active NFA states—one per pattern position—the single bitwise operation simultaneously advances all states. Wildcard positions never block a transition, since their bits are set for every letter in B.

Word-size limitation. Our implementation follows the SMART `sbndm2` convention and uses 32-bit words, so the NFA window covers at most $w = 32$ positions. For $m \leq 32$ the entire pattern fits in one word, every position is encoded in the NFA—solid positions as exact matches and wildcard positions as universal matches—and the check is both necessary and sufficient for partial-word matching within the window—yielding zero false matches at the Stage II boundary. For $m > 32$, the NFA verifies positions 1 to 32, while the remaining positions 33 to m are verified explicitly by checking, for each solid position $j > 32$ in the pattern, whether $t_\diamond[i+j-1] \approx p_\diamond[j]$.

SIMD-Accelerated Scanning. Modern processors support *Single Instruction, Multiple Data* (SIMD) operations, which apply one instruction to many data elements simultaneously using wide hardware registers. We exploit this using AVX-512 instructions, which operate on 512-bit (64-byte) registers. Instead of comparing one text position at a time, a single SIMD instruction compares 64 consecutive text positions against one pattern letter in one CPU cycle. Using a 4-letter solid anchor ($|A| = 4$) across 64 text positions per cycle gives Stage I massive scanning throughput. The trade-off is a larger Occ_{cand}, as only 4 of the m pattern positions are initially verified.

Preprocessing. The anchor $A = \{j_1, j_2, j_3, j_4\}$ consists of the first four solid positions of $p_\diamond$. For each $j_k \in A$, the letter $p_\diamond[j_k]$ is broadcast into a 512-bit register V_k, filling all 64 byte-lanes with the same value.

Searching. The text is scanned in non-overlapping blocks of 64 bytes. For a block starting at position i, the algorithm loads the 64 bytes $t_\diamond[i+j_k-1\,..\,i+j_k+62]$ into a register T_k for each anchor position j_k, and compares T_k against V_k using a single SIMD instruction. Each comparison produces a 64-bit mask M_k, where bit b is set if and only if $t_\diamond[i+b+j_k-1] = p_\diamond[j_k]$. The four masks are combined as

$$M \leftarrow M_1 \ \& \ M_2 \ \& \ M_3 \ \& \ M_4,$$

so bit b survives only if all four anchor letters match simultaneously. Every position $i+b$ for which bit b is set in M is added to Occ_{cand}.

Segment-based Filtering. This variant selects its anchor as the maximal contiguous run of solid positions in $p_\diamond$—called the *longest solid segment*. Unlike the SIMD variant, which scatters four anchor positions across the pattern, concentrating the anchor into

a single contiguous block allows the use of an exact string-matching algorithm on a fully solid substring, enabling text skipping that is not possible when wildcards are present. We use the *Horspool–BNDM* algorithm (`bmh-sbndm`) from the SMART framework, which combines the Horspool bad-character shift heuristic [21] with a BNDM bit-parallel verification step, achieving sublinear average-case behaviour in $\mathcal{O}(n/\ell)$ time for a segment of length ℓ. The strategy is most effective when $p_\diamondsuit$ has low wildcard density, because a long solid segment is highly specific and occurs rarely in the text. It degrades when high wildcard density fragments the pattern into many short solid runs, leaving no sufficiently long segment to anchor on.

Preprocessing. The algorithm scans $p_\diamondsuit$ once to find the longest contiguous run of solid letters. Let this run of length ℓ span positions $[s,\ s+\ell-1]$. The anchor is $A = \{s, s+1, \ldots, s+\ell-1\}$, and the corresponding solid substring $p_\diamondsuit[s..s+\ell-1]$ is extracted as the exact-match target. The Horspool bad-character shift table for this substring is built by the SMART implementation on each call; no shift table is stored in the pattern metadata.

Searching. Horspool–BNDM is applied to locate all occurrences of $p_\diamondsuit[s..s+\ell-1]$ in $t_\diamondsuit$. Each occurrence at text position q implies a candidate pattern start at $j = q-s+1$, provided $j \geq 1$ and $j+m-1 \leq n$. All such positions are added to Occ_{cand} and passed to Stage II. Note that Horspool–BNDM is applied to the solid segment only, not to the full wildcard pattern; applying it directly to $p_\diamondsuit$ would yield false negatives because wildcard positions cannot contribute valid shift information.

5 Experimental Results

In this section, we present a rigorous performance evaluation of the three heuristic approaches introduced in Section 4. We further compare these methods against the FFT-based Clifford–Clifford implementation of the two-phase solution. Additionally, we implemented the algorithm of Iliopoulos and Radoszewski [24] for completeness; however, its execution time proved prohibitively high, and we therefore omit it from the comparative results.[2]

In bioinformatics and text-mining scenarios, the reference text is solid, while the search pattern contains indeterminate letters (e.g., genetic mutations, read errors, or wildcards). To systematically simulate this, we synthesized our pattern sets as follows: For a given pattern of length m and an indeterminate letter density $dens$, we extract a uniformly random length-m solid substring of the text. Then, we select $dens \times m$ (D_2) distinct positions uniformly at random within the pattern and replace them with wildcards (represented by a bitmask with all bits set to 1) for Stage I and Stage II processing. To actively benchmark Stage III, each wildcard position is internally mapped to a synthesized indeterminate set containing a random combination of 2 to 4 letters from the alphabet. For each parameter configuration, we generated a set of 1000 unique patterns using a single randomized draw, and report the total elapsed time summed over all 1000 patterns.

Our evaluation isolates the computational bottlenecks of indeterminate string matching by recording the following metrics for each variant:

1. *Stage I Time:* The time elapsed to identify the candidate match set Occ_{cand}.
2. *Stage II Time:* The time elapsed to verify each Occ_{cand} to produce the set $Occ_\diamondsuit$.

[2] To facilitate reproducibility, the source code and datasets for our implementations are publicly available at `https://github.com/Hamed-Hasibi/indet_pattern_matching`.

3. *Stage III Time:* The time elapsed to verify indeterminate letter matches (Algorithm 1), producing the true match set Occ, while eliminating false matches ($|Occ_{\diamond}| - |Occ|$).

All time metrics in Tables 1, 2 and 3 and in Figure 1 represent the *total elapsed time (ms)* over 1000 unique patterns. The false match counters denote the *average* number of false positive candidates eliminated per pattern during that specific stage.

5.1 Performance Results and Discussion

The evaluation was performed on a server equipped with dual 16-core Intel Xeon Gold 6426Y processors (4.1 GHz) and 250 GB of RAM, specifically leveraging AVX-512 vectorization for the SIMD-accelerated variants. Each configuration was evaluated using 1000 patterns, and we tested four primary configurations of pattern length and wildcard density ($dens$):

- *Small*: $m = 16, dens = 12.5\%$ (2 indeterminate letters)
- *Standard (Std)*: $m = 256, dens \approx 10\%$ (25 indeterminate letters).
- *High-Density (Hi-Dens)*: $m = 1000, dens = 50\%$ (500 indeterminate letters).
- *Extreme-Density (Extr-Dens)*: $m = 1000, dens = 90\%$ (900 indeterminate letters).

We utilize the following DNA and protein datasets in FASTA format:

- *DNA* ($\sigma = 4$): The complete genome of *Escherichia coli* K-12 substr. MG1655, obtained from NCBI RefSeq (Accession `NC_000913.3`). The sequence contains 4,641,652 base pairs.
- *Protein* ($\sigma \approx 20$): The canonical proteome of *Homo sapiens*, obtained from UniProt (Reference Proteome `UP000005640, taxon 9606`). The file contains one canonical isoform per gene (≈20,400 proteins, concatenated length ≈11M amino acids).

Performance on DNA Sequences The results shown in Table 1 highlight a clear trade-off between Stage I search speed and Stage II verification overhead. The *SIMD-Accelerated* algorithm maintained a consistent Stage I runtime of approximately 795 ms regardless of pattern density, demonstrating the efficiency of AVX-512 vectorization. However, this speed comes at the cost of high false matches generated by Stage I, generating roughly 20,000 false matches per pattern in DNA.

Conversely, the *Segment-based* filter showed a severe performance drop at extreme densities (see Table 1). At $dens = 90\%$, the Stage I false-match count exceeds 162,000 candidates per pattern, causing Stage II verification time to increase to 2282.2 ms—an increase of over 5000× relative to the standard-density Stage II time of 0.4 ms. This confirms that structural partitioning is highly effective at standard densities but degrades severely as wildcard density approaches 90%, a trend illustrated in Figure 1(a).

The *Modified SBNDM2* algorithm exhibits an analogous density-dependent slowdown. Its Stage I runtime grows from 1657.2 ms at standard density to 9489.7 ms at extreme density (Table 1)—a roughly 5.7× increase. The root cause is the 32-bit word-size limitation of the NFA: for patterns longer than $w = 32$ symbols, positions 33 to m must be verified letter-by-letter outside the NFA loop. As wildcard density rises, more wildcard positions fall within the NFA window, suppressing skip opportunities and forcing the algorithm to enter the explicit suffix-verification path on nearly

Table 1. Benchmark results on the DNA (*E. coli*) dataset. All time metrics represent the *total elapsed time (ms)* over 1000 unique patterns. The false match count represents the *average* number of false positive candidates eliminated per pattern during that specific stage.

Algorithm	**Small**	**Std**	**Hi-Dens**	**Extr-Dens**
Stage I: Candidate Generation Time (ms)				
Modified SBNDM2	3162.3	1657.2	2820.3	9489.7
SIMD-Accelerated	798.4	797.5	797.8	792.7
Segment-based	4548.4	2037.4	4451.5	7630.1
Stage II: Solid Letter Verification Time (ms) / Stage I False Matches				
Modified SBNDM2	0.1 / 0.0	0.4 / 0.0	0.6 / 0.0	0.3 / 0.0
SIMD-Accelerated	205.7 / 20677.2	203.1 / 20472.7	256.0 / 19624.8	226.0 / 18652.9
Segment-based	4.1 / 373.9	0.4 / ≈ 0.0	5.4 / 101.4	2282.2 / 163676.1
Stage III: Indeterminate Verification Time (ms) / Stage II False Matches				
Modified SBNDM2	0.2 / 0.80	0.4 / 1.03	0.4 / 1.00	0.4 / 0.62
SIMD-Accelerated	0.4 / 0.80	0.3 / 1.03	0.2 / 1.00	0.3 / 1.01
Segment-based	0.2 / 0.80	0.3 / 1.03	0.2 / 1.00	0.2 / 1.01
Total Elapsed Time: Stage I + II + III (ms)				
Modified SBNDM2	3162.6	1658.0	2821.3	9490.4
SIMD-Accelerated	1004.5	1000.9	1054.0	1019.0
Segment-based	4552.7	2038.1	4457.1	9912.5

every window. The result is a near-linear increase in Stage I runtime with density, mirroring the Segment-based degradation. Notably, because the NFA accounts for *all* solid positions within its window, Stage II incurs zero false matches and remains negligible (≤ 0.6 ms) across all configurations—a precision advantage that does not, however, compensate for the slow scan rate when compared against SIMD (Table 3 and Figure 1(a)).

Performance on Protein Sequences The "Alphabet Effect" is clearly visible when comparing Tables 1 and 2, revealing a counter-intuitive anomaly: despite the protein text being $\approx 2.4\times$ longer than the DNA text, Stage I SIMD scanning completed in approximately 260 ms for Protein compared to 795 ms for DNA. This discrepancy is entirely driven by the memory-write bottleneck of the candidate buffer. For example, for strings over the DNA alphabet ($\sigma = 4$), a 4-letter solid anchor matches by chance with a probability of 1/256, causing the SIMD inner loop to trigger and write to memory roughly 18,000 times per pattern. However, for strings over the Protein alphabet ($\sigma \approx 20$), this probability drops to $1/160,000$, resulting in an average of only 51 writes per pattern. This virtual elimination of write overhead allows the SIMD loop on the protein dataset to operate much closer to the hardware's true memory bandwidth limits. Despite the protein text being $\approx 2.4\times$ longer than the DNA text, all algorithms ran $3\times$ to $4\times$ faster on it, corresponding to a roughly $10\times$ improvement in per-character throughput attributable to the larger alphabet. Most notably, the SIMD false matching rate dropped from 18652 to 51 per pattern. This higher specificity allows the Segment-based filter to remain competitive even at extreme densities, where its verification time in Protein (51.8 ms) is significantly lower than its DNA counterpart (2282.2 ms).

Importantly, the Stage III performance demonstrates the mathematically consistent properties of the framework. Because Stage III operates strictly on $Occ_\diamond$ (the

Table 2. Benchmark results for 1000 patterns on the Protein (*H. sapiens*) dataset.

Algorithm	**Small**	**Std**	**Hi-Dens**	**Extr-Dens**
Stage I: Candidate Generation Time (ms)				
Modified SBNDM2	772.1	486.1	1086.2	4714.7
SIMD-Accelerated	262.6	256.5	265.6	275.3
Segment-based	556.9	314.7	543.7	1771.0
Stage II: Solid Letter Verification Time (ms) / Stage I False Matches				
Modified SBNDM2	0.1 / 0.0	0.4 / 0.0	4.2 / 0.0	1.5 / 0.0
SIMD-Accelerated	0.7 / 58.0	1.0 / 51.6	5.1 / 52.8	3.6 / 51.1
Segment-based	0.1 / 0.2	0.4 / ≈ 0.0	4.3 / 0.1	51.8 / 5308.9
Stage III: Indeterminate Verification Time (ms) / Stage II False Matches				
Modified SBNDM2	0.2 / 1.08	0.2 / 1.09	0.2 / 1.00	0.2 / 0.88
SIMD-Accelerated	0.1 / 1.08	0.2 / 1.09	0.2 / 1.00	0.2 / 1.00
Segment-based	0.2 / 1.08	0.2 / 1.09	0.2 / 1.00	0.2 / 1.00
Total Elapsed Time: Stage I + II + III (ms)				
Modified SBNDM2	772.4	486.7	1090.6	4716.4
SIMD-Accelerated	263.4	257.7	270.9	279.1
Segment-based	557.2	315.3	548.2	1823.0

purified partial word candidates), the verification time is nearly instantaneous (sub-millisecond) for almost all configurations. The uniformly low False matching rates (~ 1.0) across all three algorithms in Stage III confirm that despite generating vastly different candidate sets in Stage I, the algorithms all perfectly converge to the identical set of valid matches by the conclusion of Stage II.

In the DNA dataset at extreme density ($dens = 90\%$), the SIMD-Accelerated variant generated an overwhelming 18,652.9 false matches per pattern, incurring a Stage II verification penalty of 226 ms. By contrast, Modified SBNDM2 achieved perfect precision with 0.0 false matches and only 0.3 ms of verification time. However, because the AVX-512 vectorization scans the text in 64-byte blocks without branching, its Stage I search completed in just 792.7 ms, compared to 9489.7 ms for the table-driven SBNDM2. Consequently, the *total* running time for SIMD (1019 ms) was around 9× faster than the most precise algorithm (9490.4 ms). This trend is absolute across all configurations in both datasets. We therefore conclude that for indeterminate string matching on modern architectures, maximizing raw scanning throughput (via SIMD) is vastly superior to minimizing false matches (via structural partitioning or bit-parallel automata). The computational cost of verifying false matches in Stage II is trivial compared to the cycle overhead required by complex Stage I filtering logic.

We demonstrate the runtime differences between the three implementations and one of the fastest FFT-based wildcard pattern matching implementations. To rigorously benchmark these three implementations, we implemented the Clifford–Clifford $\mathcal{O}(n \log m)$ deterministic wildcard matching algorithm using the highly optimized `FFTW3`[3] library. Since the FFT is considerably slow, we ran all four algorithms on a reduced set of 100 patterns (rather than 1000) and report total elapsed times in Table 3. Note that the text FFT is only computed once across all patterns and does

[3] https://github.com/FFTW/fftw3/tree/master

not have the Stage-I heuristic. `FFTW3` library is coded using Clifford and Clifford $\mathcal{O}(n \log m)$ time complexity.

Table 3. Total elapsed time (ms) for 100 patterns combining all stages, on the DNA (*E. coli*) and Protein (*H. sapiens*) datasets.

Algorithm	DNA (*E. coli*)				Protein (*H. sapiens*)			
	Small	**Std**	**Hi-Dens**	**Extr-Dens**	**Small**	**Std**	**Hi-Dens**	**Extr-Dens**
Modified SBNDM2	320.2	161.0	296.9	932.1	269.4	160.6	353.1	1410.2
SIMD-Accelerated	106.2	110.3	107.6	104.9	97.8	95.7	98.6	92.5
Segment-based	771.6	365.9	752.3	1738.5	679.9	307.5	655.5	1982.2
FFT (Clifford-Cl.)	4471.9	7729.1	9392.6	9155.8	11043.1	19089.5	23142.2	22541.5

As shown in Table 3, the FFT-based Clifford–Clifford implementation is dramatically slower than all three pipeline variants in every configuration. On DNA, FFT is 42× to 87× slower than SIMD-Accelerated; on protein, the gap widens to 113× to 244× despite FFT's alphabet-independence advantage. This counter-intuitive widening is explained by the longer protein text (11M amino acids vs. 4.6M base pairs): FFT's $\mathcal{O}(n \log m)$ runtime scales with text length, while SIMD's runtime is essentially constant across densities and benefits from cache-friendly streaming access patterns. These results justify our decomposition of Phase I into a heuristic Stage I plus a verification Stage II in the practical pipeline—raw FFT throughput is too low to be competitive even before Stage II/III work is added.

Figure 1 presents the total elapsed time across three parameters on both datasets. Subplots (a) and (d) confirm that SIMD and FFT maintain a near-flat runtime as wildcard density increases, while Modified SBNDM2 and Segment-based filtering degrade monotonically. Subplots (b) and (e) show that FFT scaling with pattern length is severe—exceeding 10^5 ms at $m{=}10^6$—whereas all three pipeline variants remain stable. Subplots (c) and (f) demonstrate that all algorithms scale linearly with the number of patterns, with SIMD consistently achieving the lowest slope. Across all six subplots, SIMD-Accelerated dominates, and FFT lags behind by one to three orders of magnitude.

6 Conclusion and Future Work

In this paper, we presented a highly efficient approach to the indeterminate pattern matching problem, bridging the gap between optimal theoretical bounds and modern hardware capabilities. We first introduced a simple two-phase theoretical algorithm that computes all occurrences in $\mathcal{O}(n \log m + |Occ_\diamond| \min(m, D_1 + D_2))$ time. This theoretical foundation eliminates the exponential dependence on alphabet size found in the $\mathcal{O}(2^\sigma n \log m)$-time algorithm of Iliopoulos and Radoszewski [24, Lemma 17].

We expand the two-phase approach into an adaptable fast and practical three-stage pipeline and evaluate three distinct heuristic strategies for Stage I: a table-driven Modified SBNDM2 automaton, SIMD-Accelerated scanning, and Segment-based filtering. Our comprehensive experimental evaluation on DNA and protein datasets reveals a counterintuitive but definitive conclusion: on modern architectures, maximizing raw scanning throughput via AVX-512 vectorization fundamentally outperforms complex algorithmic filtering. Despite generating a significantly higher volume

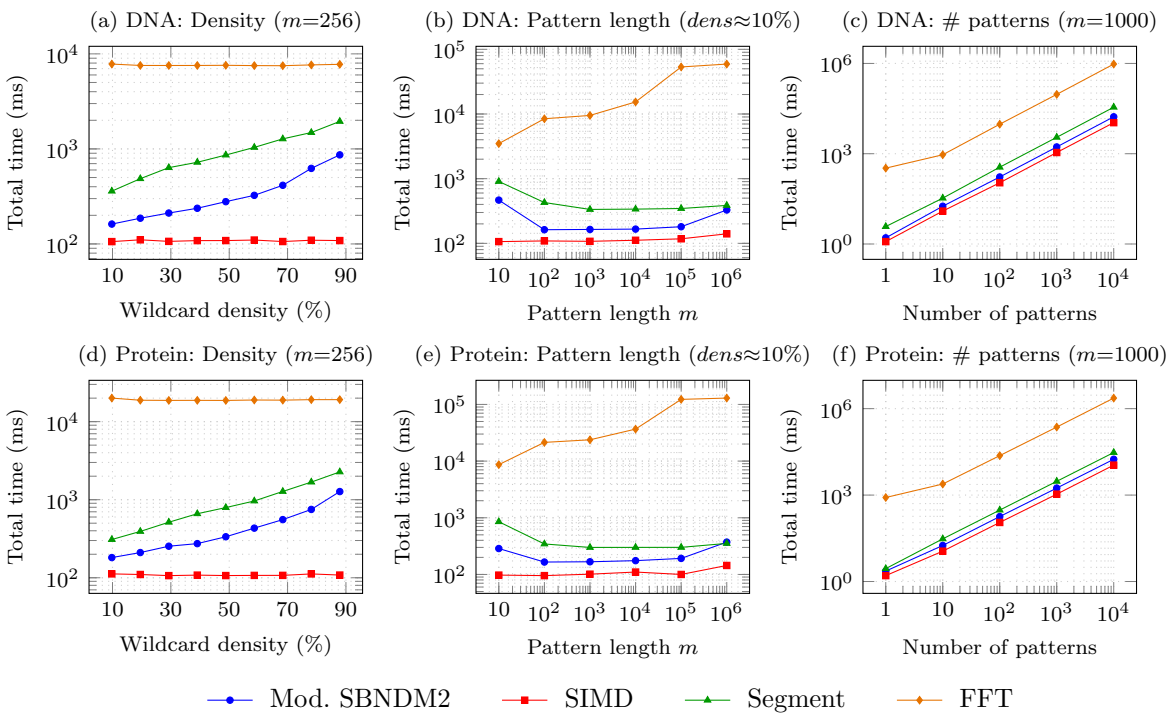

Figure 1. Total elapsed time (Stage I + II + III, ms over 100 patterns) across three experiments on DNA (*E. coli*, top row) and protein (*H. sapiens*, bottom row). Columns show: (a,d) wildcard-density changes at fixed pattern length m=256; (b,e) pattern-length changes with *dens*≈10%; (c,f) pattern-count changes at fixed m=1000, *dens*=10%. Both axes are log-scaled in (b),(c),(e),(f); the y-axis is log-scaled in (a),(d). A shared legend below the subplots applies to all six plots.

of false candidates, the SIMD-accelerated pipeline easily absorbed the Stage II verification overhead, achieving the fastest end-to-end running times and outperforming traditional FFT-based convolution methods by orders of magnitude.

In future work, we plan to compare our approach with existing indeterminate pattern matching methods that have available implementations, including the Degenerate Burrows–Wheeler Transform approach of Daykin et al. [12], the k-mismatch approach of Crochemore et al. [10], and the KMP/BM-style solutions of Dehghani et al. [13,29]. An adaptive algorithm could analyze wildcard density at runtime and switch seamlessly between SIMD scanning for dense wildcards and segment-based filtering when a sufficiently long solid island is detected. Extending this three-stage verification paradigm to more complex degenerate models, such as Elastic Degenerate (ED) strings [22], or adapting it for offline, index-based pattern matching over large pangenomic databases also remains a compelling direction for future research.

Acknowledgements: We thank Hossein Dehghani and Zehong Wang for useful discussions on the two-phase algorithm.

References

1. K. Abrahamson: *Generalized string matching.* SIAM journal on Computing, 16(6) 1987, pp. 1039–1051.
2. P. Antoniou, M. Crochemore, C. S. Iliopoulos, I. Jayasekera, and G. M. Landau: *Conservative string covering of indeterminate strings.*, in Proceedings of the Prague Stringology Conference, 2008, pp. 108–115.

3. G. Bathie, P. Charalampopoulos, and T. Starikovskaya: *Pattern Matching with Mismatches and Wildcards*, in 32nd Annual European Symposium on Algorithms (ESA 2024), vol. 308 of Leibniz International Proceedings in Informatics (LIPIcs), 2024, pp. 20:1–20:15.
4. P. Bille, I. L. Gørtz, H. W. Vildhøj, and S. Vind: *String indexing for patterns with wildcards*. Theory of Computing Systems, 55 2014, pp. 41–60.
5. R. S. Boyer and J. S. Moore: *A fast string searching algorithm*. Communications of the ACM, 20(10) 1977, pp. 762–772.
6. G. Chen, X. Wu, X. Zhu, A. N. Arslan, and Y. He: *Efficient string matching with wildcards and length constraints*. Knowledge and information systems, 10 2006, pp. 399–419.
7. P. Clifford and R. Clifford: *Simple deterministic wildcard matching*. Information Processing Letters, 101(2) 2007, pp. 53–54.
8. R. Cole, L.-A. Gottlieb, and M. Lewenstein: *Dictionary matching and indexing with errors and don't cares*, in Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, 2004, pp. 91–100.
9. R. Cole and R. Hariharan: *Verifying candidate matches in sparse and wildcard matching*, in Proceedings of the thiry-fourth annual ACM symposium on Theory of computing, 2002, pp. 592–601.
10. M. Crochemore, C. S. Iliopoulos, R. Kundu, M. Mohamed, and F. Vayani: *Linear algorithm for conservative degenerate pattern matching*. Engineering Applications of Artificial Intelligence, 51 2016, pp. 109–114.
11. M. Crochemore and W. Rytter: *Text algorithms*, Maxime Crochemore, 1994.
12. J. W. Daykin, R. Groult, Y. Guesnet, T. Lecroq, A. Lefebvre, M. Léonard, L. Mouchard, É. Prieur-Gaston, and B. Watson: *Efficient pattern matching in degenerate strings with the burrows–wheeler transform*. Information Processing Letters, 147 2019, pp. 82–87.
13. H. Dehghani, T. Lecroq, N. Mhaskar, and W. Smyth: *Practical KMP/BM style pattern-matching on indeterminate strings*. Discrete Applied Mathematics, 370 2025, pp. 22–33.
14. S. Faro, T. Lecroq, S. Borzì, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference, 2016, pp. 99–111.
15. M. Fischer and M. Paterson: *String matching and other products*, in Complexity of Computation, R. Karp, ed., American Mathematical Society, 1974, pp. 113–125.
16. P. Gawrychowski, S. Ghazawi, and G. M. Landau: *On indeterminate strings matching*, in 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020), vol. 161 of Leibniz International Proceedings in Informatics (LIPIcs), 2020, pp. 14:1–14:14.
17. P. Gawrychowski, S. Ghazawi, and G. M. Landau: *On the complexity of indeterminate strings matching*. Theoretical Computer Science, 1067 2026, p. 115771.
18. R. Grossi, C. S. Iliopoulos, C. Liu, N. Pisanti, S. P. Pissis, A. Retha, G. Rosone, F. Vayani, and L. Versari: *On-Line Pattern Matching on Similar Texts*, in 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), vol. 78 of Leibniz International Proceedings in Informatics (LIPIcs), 2017, pp. 9:1–9:14.
19. J. Holub, W. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings*. Journal of Discrete Algorithms, 6(1) 2008, pp. 37–50, Selected papers from AWOCA 2005.
20. J. Holub, W. F. Smyth, and S. Wang: *Hybrid pattern-matching algorithms on indeterminate strings*, in London Algorithmics and Stringology, J. W. Daykin, M. Mohamed, and K. Steinhofel, eds., King's College Texts in Algorithmics, 2006, pp. 115–133.
21. R. N. Horspool: *Practical fast searching in strings*. Software: Practice and Experience, 10(6) 1980, pp. 501–506.
22. C. S. Iliopoulos, R. Kundu, and S. P. Pissis: *Efficient pattern matching in elastic-degenerate strings*. Information and Computation, 279 2021, p. 104616.
23. C. S. Iliopoulos, L. Mouchard, and M. S. Rahman: *A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching*. Mathematics in Computer Science, 1(4) 2008, pp. 557–569.
24. C. S. Iliopoulos and J. Radoszewski: *Truly subquadratic-time extension queries and periodicity detection in strings with uncertainties*, in 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
25. P. Indyk: *Faster algorithms for string matching problems: Matching the convolution bound*, in Proceedings 39th Annual Symposium on Foundations of Computer Science, IEEE, 1998, pp. 166–173.

26. A. Kalai: *Efficient pattern-matching with don't cares*, in Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, 2002, pp. 655–656.
27. D. E. Knuth, J. H. Morris, and V. R. Pratt: *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2) 1977, pp. 323–350.
28. F. A. Louza, N. Mhaskar, and W. Smyth: *A new approach to regular & indeterminate strings.* Theoretical Computer Science, 854 2021, pp. 105–115.
29. N. Mhaskar and W. F. Smyth: *Simple kmp pattern-matching on indeterminate strings*, in Proceedings of the Prague Stringology Conference, 2020, pp. 125–133.
30. A. A. Naeini, A.-P. Mobed, M. Seddighin, and S. Seddighin: *Dynamic pattern matching with wildcards.* arXiv preprint arXiv:2601.16182, 2026.
31. R. Y. Pinter: *Efficient string matching with don't-care patterns*, in Combinatorial Algorithms on Words, Springer, 1985, pp. 11–29.
32. M. S. Rahman and C. S. Iliopoulos: *Pattern matching algorithms with don't cares*, in Conference on Current Trends in Theory and Practice of Informatics (SOFSEM), 2007.
33. M. S. Rahman, C. S. Iliopoulos, I. Lee, M. Mohamed, and W. F. Smyth: *Finding patterns with variable length gaps or don't cares*, in Computing and Combinatorics: 12th Annual International Conference, COCOON 2006, 2006. Proceedings 12, 2006, pp. 146–155.
34. B. Smyth: *Computing patterns in strings*, Pearson Education, 2003.
35. W. F. Smyth and S. Wang: *An adaptive hybrid pattern-matching algorithm on indeterminate strings.* IJFCS, 20(6) 2009, pp. 985–1004.

On the Additive Sensitivity of CDAWG-grammars

Hiroto Fujimaru[1] ⋆ and Shunsuke Inenaga[2] ⋆⋆

[1] Department of Information Science and Technology, Kyushu University, Japan
fujimaru.hiroto.134@s.kyushu-u.ac.jp
[2] Department of Informatics, Kyushu University, Japan
inenaga.shunsuke.380@m.kyushu-u.ac.jp

Abstract. The *compact directed acyclic word graph* (*CDAWG*) [Blumer et al. 1987] of a string is the minimal compact automaton that recognizes all the suffixes of the string. CDAWGs can be used for various string tasks including text pattern searching, data compression, and pattern discovery. The *CDAWG-grammar* [Belazzougui & Cunial 2017] is a grammar-based text compression based on the CDAWG, which allows for representing the CDAWG in $O(\mathsf{e})$ space without storing the string, where e denotes the number of CDAWG edges. Let g be the size of the CDAWG-grammar for the input string T. We exhibit an infinite family of strings for which the CDAWG-grammar size increases by $3g-21$ after one edit, and we show that the worst-case additive sensitivity is upper bounded by $8g+4$.

1 Introduction

The *compact directed acyclic word graph* (*CDAWG*) [5] of a string is the minimal compact automaton that recognizes all the suffixes of the string. CDAWGs are known to be useful for various string tasks including text pattern searching [7,10], data compression [4,3,13], and pattern discovery [14].

CDAWGs can be obtained by merging isomorphic subtrees of suffix trees [5,4,3]. Due to this nature, as in the case of suffix trees of which each edge string label x is represented by a pair (i, j) of positions satisfying $T[i..j] = x$, classical implementations of CDAWGs also require $\Theta(n)$ space to explicitly store a copy of the string T being indexed, where $n = |T|$. This significantly limits the size of strings on which the CDAWGs are built. Still, the number e of the edges in the CDAWG tends to be small, for some highly repetitive strings, in particular, $\mathsf{e} = O(\log n)$ holds for the standard Sturmian words and Thue-Morse words [12,2,11].

The task of representing the CDAWG with $O(\mathsf{e})$ space was first achieved by Belazzougui and Cunial [4,3], via a connection from CDAWGs to *grammar-based compressions*. They observed that the reversed CDAWG induces a context-free grammar (CFG) that only generates the input string T, which is hereafter named as the *CDAWG-grammar*. By augmenting the CDAWG-grammar with a constant-time level ancestor data structure, the CDAWG can be stored in $O(\mathsf{e})$ space while retaining the ability to perform optimal $O(m + occ)$-time pattern matching, where m is the pattern length and occ is the number of occurrences to report [4,3]. While the CDAWG-grammar can easily be obtained via the graph topology of the corresponding CDAWG, it is also possible to build the CDAWG-grammar *directly* from suffix-array-based structures [6], without the need of explicitly building the CDAWG.

⋆ Supported by JST BOOST Grant Number JPMJBS2406.
⋆⋆ Supported by JSPS KAKENHI Grant Numbers JP23K24808 and JP23K18466.

Hiroto Fujimaru, Shunsuke Inenaga: On the Additive Sensitivity of CDAWG-grammars, pp. 43–57.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

The focus of this paper is to analyze the advantage of CDAWG-grammar in terms of *sensitivity*: The sensitivity of a repetitiveness measure c asks how much the measure size increases when a single-character edit operation is performed on the input string [1], and thus the sensitivity allows one to evaluate the robustness of the measure/compressor against errors/edits. Recent works have revealed the sensitivity of CDAWG size e in the case of left-end edit operations [8] as well as in the general case [9]. As for the CDAWG-grammar size g, it is known that $\mathsf{g} = \mathsf{e} - \mathrm{v}^{(1)}$ holds [4,3], where $\mathrm{v}^{(1)}$ denotes the number of CDAWG nodes of in-degree one. To obtain non-trivial bounds for the worst-case additive sensitivity of the CDAWG-grammar size g, we present new properties of $\mathrm{v}^{(1)}$ which were not previously well understood.

In this paper, we first provide an infinite family of strings such that $\mathsf{g}(T') - \mathsf{g}(T) = 3\mathsf{g} - 21$ for the one-character edit $T \mapsto T'$. After presenting a straightforward upper bound of $15\mathsf{g} + 4$, we give a non-trivial tighter upper bound of $8\mathsf{g} + 4$ based on combinatorial properties of CDAWGs and CDAWG-grammars. Thus, on the infinite sequence of input lengths realized by our construction, the lower and upper bounds are tight up to a constant factor.

2 Preliminaries

Strings. Let Σ be an *alphabet* of size σ. An element of Σ^* is called a *string*. For a string $T \in \Sigma^*$, the length of T is denoted by $|T|$. The *empty string*, denoted by ε, is the string of length 0. Let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. If $T = uvw$, then u, v, and w are called a *prefix*, *substring*, and *suffix* of T, respectively. The sets of prefixes, substrings, and suffixes of string T are denoted by $\mathsf{Prefix}(T)$, $\mathsf{Substr}(T)$, and $\mathsf{Suffix}(T)$, respectively. For a string T of length n, $T[i]$ denotes the ith character of T for $1 \le i \le n$, and $T[i..j] = T[i] \cdots T[j]$ denotes the substring of T that begins at position i and ends at position j on T for $1 \le i \le j \le n$. For $0 \le k \le |T|$, let $\varepsilon = T[k+1..k]$.

Maximal substrings. For two strings u and T, let $\mathsf{BegPos}(u, T) = \{i \mid T[i..i+|u|-1] = u\}$ and $\mathsf{EndPos}(u, T) = \{i \mid T[i-|u|+1..i] = u\}$ denote the sets of beginning and ending positions of u in T, respectively. For any substrings $u, v \in \mathsf{Substr}(T)$ of a string T, define $u \equiv^{\mathrm{L}}_T v \Leftrightarrow \mathsf{EndPos}(u, T) = \mathsf{EndPos}(v, T)$, and $u \equiv^{\mathrm{R}}_T v \Leftrightarrow \mathsf{BegPos}(u, T) = \mathsf{BegPos}(v, T)$. A substring $x \in \mathsf{Substr}(T)$ of T is said to be *left-maximal* in T if $ax, bx \in \mathsf{Substr}(T)$ for some two distinct characters $a, b \in \Sigma$, or $x \in \mathsf{Prefix}(T)$. Similarly, x is said to be *right-maximal* in T if $xa, xb \in \mathsf{Substr}(T)$ for some two distinct characters $a, b \in \Sigma$, or $x \in \mathsf{Suffix}(T)$. Let $\mathsf{LeftM}(T)$ and $\mathsf{RightM}(T)$ denote the sets of left-maximal and right-maximal substrings in T, respectively. It is known [5] that $x \in \mathsf{LeftM}(T)$ (resp. $x \in \mathsf{RightM}(T)$) iff x is the longest member of the equivalence class under $\equiv^{\mathrm{L}}_T$ (resp. under $\equiv^{\mathrm{R}}_T$).

Let $\mathsf{M}(T) = \mathsf{LeftM}(T) \cap \mathsf{RightM}(T)$. Any element of $\mathsf{M}(T)$ is said to be *maximal* in T.

A string x is said to be a repeat in a string T if $|\mathsf{BegPos}(x, T)| = |\mathsf{EndPos}(x, T)| \ge 2$. Let $\mathsf{MR}(T) = \mathsf{M}(T) \setminus \{T\}$ denote the set of *maximal repeats* in T.

For any substring S of a string T, we define its *right-representative* as $\mathsf{rexp}_T(S) = S\beta$, where $\beta \in \Sigma^*$ is the shortest string such that $S\beta$ is right-maximal in T.

CDAWGs. The *suffix tree* [15] of a string T is a rooted tree such that each edge is labeled by a non-empty substring of T and the path from the root to each node v represents a right-maximal substring in $\mathsf{RightM}(T)$. By assuming a unique end-marker \$ at the right-end of T, every internal node of the suffix tree of T has at least two children, and there is a one-to-one correspondence between the leaves and the

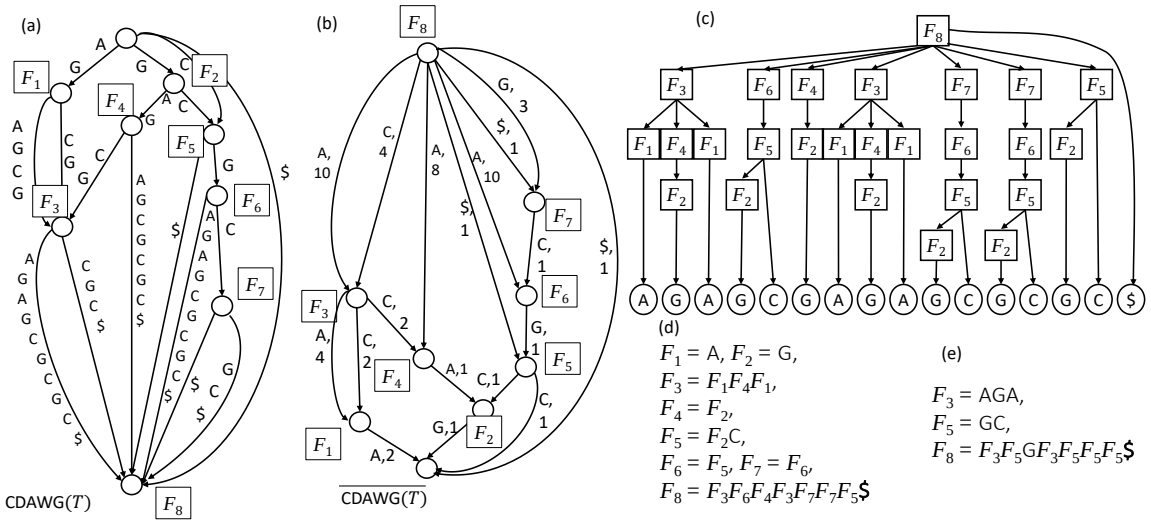

Figure 1. (a) $\mathsf{CDAWG}(T)$ of string T = AGAGCGAGAGCGCGC\$ for which $\mathsf{M}(T) = \{\varepsilon, \mathtt{G}, \mathtt{GC}, \mathtt{AG}, \mathtt{GAG}, \mathtt{GCG}, \mathtt{GCGC}, \mathtt{AGAGCG}, T\}$. The number of right-extensions of $\mathsf{CDAWG}(T)$ is the number $\mathsf{e}(T)$ of edges, which is 18 in this example. (b) The reversed DAG $\overline{\mathsf{CDAWG}(T)}$, where each edge is labeled by the initial character and the length of the edge's original label. (c) The derivation tree $\mathcal{T}(T)$ obtained by unfolding $\overline{\mathsf{CDAWG}(T)}$. (d) The grammar rules obtained from $\mathcal{T}(T)$. (e) The resulting grammar $\mathsf{G}_{\mathsf{CDAWG}}(T)$ without redundant rules. This grammar size is 13.

suffixes of T. The *compact directed acyclic word graph* (*CDAWG*) [5] of a string T, denoted $\mathsf{CDAWG}(T) = (\mathsf{V}_T, \mathsf{E}_T)$, is an edge-labeled smallest DAG that is obtained by merging isomorphic subtrees of the suffix tree for T. See the left diagram of Fig. 1 for a concrete example of CDAWGs. It is known that there is a one-to-one correspondence between the nodes of $\mathsf{CDAWG}(T)$ and the maximal substrings in $\mathsf{M}(T)$, such that the longest string represented by each node of $\mathsf{CDAWG}(T)$ is a maximal substring. In what follows, we identify the nodes of $\mathsf{CDAWG}(T)$ with the maximal substrings in $\mathsf{M}(T)$. In particular, we identify the non-sink nodes of $\mathsf{CDAWG}(T)$ with the maximal repeats in $\mathsf{MR}(T)$. For any $x \in \mathsf{MR}(T)$, $\mathsf{d}_T(x)$ denotes the out-degree of the node x in $\mathsf{CDAWG}(T)$. Similar to suffix trees, every internal node x of the $\mathsf{CDAWG}(T)$ has at least two children if T terminates with \$. Hence, the following fact holds.

Fact 1 *If T terminates with* \$, $\mathsf{d}_T(x) \geq 2$ *for any* $x \in \mathsf{MR}(T)$.

The *size* of $\mathsf{CDAWG}(T) = (\mathsf{V}_T, \mathsf{E}_T)$ for a string T of length n is the number $\mathsf{e}(T) = |\mathsf{E}_T|$ of edges in $\mathsf{CDAWG}(T)$, which is also referred to as the number of right-extensions of maximal repeats in T. Formally, a character $a \in \Sigma$ is a right-extension of a substring x in T if $xa \in \mathsf{Substr}(T)$; the distinct right-extensions of a CDAWG node are represented by its outgoing edges.

CDAWG-grammars. A *context-free grammar* (CFG) $\mathcal{G}$ which generates only a 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 $\mathcal{G}$.

Suppose that the input string T terminates with a unique end-marker \$. The *CDAWG grammar* [4,3] of T, denoted $\mathsf{G}_{\mathsf{CDAWG}}(T)$, is a grammar compression for T that is built on $\mathsf{CDAWG}(T)$ as follows (see also Fig. 1): Let $\overline{\mathsf{CDAWG}(T)}$ denote the DAG obtained by reversing all edges of $\mathsf{CDAWG}(T)$. A critical observation is that, since $\mathsf{CDAWG}(T)$ is a DAG obtained by merging subtrees of the suffix tree for T, the source-to-sink paths in $\mathsf{CDAWG}(T)$ have a one-to-one correspondence with the suffixes of T, and so do the corresponding paths in $\overline{\mathsf{CDAWG}(T)}$. We label each

non-source node v in $\mathsf{CDAWG}(T)$ by a non-terminal F_i for $1 \leq i \leq |\mathsf{V}_T| - 1$ in an arbitrary topological order, and let p_k denote the path of $\mathsf{CDAWG}(T)$ that represents the suffix $T[k..|T|]$ starting at position k in T. Then, we build a tree $\mathcal{T}(T)$ by inserting, for each $1 \leq k \leq |T|$, the reversed path $\overline{p_k}$ of $\overline{\mathsf{CDAWG}(T)}$ as the path of $\mathcal{T}(T)$ from the root to the kth leaf that corresponds to the kth character $T[k]$. In other words, $\mathcal{T}(T)$ is the tree obtained by unfolding $\overline{\mathsf{CDAWG}(T)}$ into a tree. Observe that $\mathcal{T}(T)$ is the derivation tree of a grammar that generates (only) T, and its DAG representation is $\overline{\mathsf{CDAWG}(T)}$. This initial grammar may contain redundant unary rules whose right-hand side consists of a single symbol, either a terminal or a non-terminal (e.g., $F_6 \to F_5$ in Diagram (c) of Fig. 1). We remove all such redundant rules, and the resulting grammar is the CDAWG grammar $\mathsf{G}_{\mathsf{CDAWG}}(T)$ for T.

Let $\mathsf{g}(T)$ denote the size of $\mathsf{G}_{\mathsf{CDAWG}}(T)$. Let $F_i \to w \in (\Sigma \cup \{F_1, \ldots, F_{i-1}\})^+$ be a production in $\mathsf{G}_{\mathsf{CDAWG}}(T)$. The length of the right-hand side of the production in $\mathsf{G}_{\mathsf{CDAWG}}(T)$ is equal to the number of outgoing edges of the node of $\overline{\mathsf{CDAWG}(T)}$ that corresponds to F_i. This is equal to the number of in-edges of the node of $\mathsf{CDAWG}(T)$. Thus, we have $\mathsf{g}(T) \leq \mathsf{e}(T)$. Recalling that the redundant rules have been removed in $\mathsf{G}_{\mathsf{CDAWG}}(T)$, we obtain the following equation with the number $\mathrm{v}^{(1)}(T)$ of nodes of in-degree 1 in $\mathsf{CDAWG}(T)$.

Fact 2 $\mathsf{g}(T) = \mathsf{e}(T) - \mathrm{v}^{(1)}(T)$.

3 Sensitivity of CDAWG-grammar g and naive bound

We define the worst-case *additive sensitivity* of the CDAWG-grammar for any string of length n that terminates with \$ as follows:

$$\begin{aligned}\mathsf{AS}(\mathsf{g}, n) &= \max_{T \in \Sigma^n, T' \in \mathcal{K}(T,1)} \{\mathsf{g}(T') - \mathsf{g}(T) \mid T[|T|] = T'[|T'|] = \$\} \\ &= \max_{T \in \Sigma^n, T' \in \mathcal{K}(T,1)} \{\mathsf{e}(T') - \mathsf{e}(T) + \mathrm{v}^{(1)}(T) - \mathrm{v}^{(1)}(T') \mid T[|T|] = T'[|T'|] = \$\} \quad (1)\end{aligned}$$

where $\mathcal{K}(T, 1)$ denotes the set of strings of edit distance 1 from T. The second equation is due to Fact 2.

It follows from Fact 1 that $2\mathrm{v}^{(1)}(T) \leq 2|\mathsf{MR}(T)| \leq \mathsf{e}(T)$. Thus, by Fact 2 we get $\mathsf{e}(T)/2 \leq \mathsf{g}(T) \leq \mathsf{e}(T)$. Hamai et al. [9] showed that $\mathsf{e}(T') \leq 8\mathsf{e}(T) + 4$ holds for any strings T and $T' \in \mathcal{K}(T, 1)$. Using these inequalities, it immediately follows that $\mathsf{g}(T') \leq \mathsf{e}(T') \leq 8\mathsf{e}(T) + 4 \leq 16\mathsf{g}(T) + 4$, which implies $\mathsf{AS}(\mathsf{g}, n) \leq 15\mathsf{g}(T) + 4$. In the following sections, we improve the constant factor of $\mathsf{AS}(\mathsf{g}, n)$ from 15 to 8.

4 Lower bound for the sensitivity of CDAWG-grammar

Below, we present a lower bound for $\mathsf{AS}(\mathsf{g}, n)$:

Theorem 1. *There exists an infinite family of strings T such that $\mathsf{g}(T') - \mathsf{g}(T) = 3\mathsf{g}(T) - 21$ with $T' \in \mathcal{K}(T, 1)$. Thus, the worst-case additive sensitivity is lower bounded by $3\mathsf{g} - 21$ for infinitely many input lengths.*

Proof. Let $T = (ab)^m c(ab)^m c\$$ for an integer $m \geq 2$, where $a, b, c, \$ \in \Sigma$ are pairwise distinct characters. Let T' be obtained from T by substituting the first c with b, namely $T' = (ab)^m b(ab)^m c\$$. Clearly, $T' \in K(T, 1)$, and both T and T' terminate with the same end-marker \$.

We first compute $\mathsf{g}(T)$. The maximal substrings of T, except for the sink T, are precisely

$$\varepsilon, \quad (ab), (ab)^2, \ldots, (ab)^{m-1}, \quad (ab)^m c.$$

Indeed, for $1 \leq r \leq m-1$, the string $(ab)^r$ is both left- and right-maximal: it has occurrences preceded by different left contexts (the beginning of T, b, or c) and occurrences followed by different right contexts a and c. Also, $(ab)^m c$ is both left- and right-maximal, since its two occurrences are preceded by the beginning of T and by c, and are followed by a and \$, respectively. No other substring is both left- and right-maximal: a substring not containing c is a factor of the alternating blocks $(ab)^m$, and only the even aligned factors $(ab)^r$ can have two right contexts; among them, $(ab)^m$ has only the right context c. A substring containing c can be left-maximal only when it is $(ab)^m c$ or T.

The right-extensions of these maximal repeats are as follows: $\mathsf{d}_T(\varepsilon) = 4$, $\mathsf{d}_T((ab)^r) = 2$ for $1 \leq r \leq m-1$, and $\mathsf{d}_T((ab)^m c) = 2$. Hence $\mathsf{e}(T) = 4 + 2(m-1) + 2 = 2m + 4$.

Moreover, the nodes of in-degree one are exactly

$$(ab)^2, (ab)^3, \ldots, (ab)^{m-1},$$

and therefore $\mathrm{v}^{(1)}(T) = m-2$. By Fact 2, $\mathsf{g}(T) = \mathsf{e}(T) - \mathrm{v}^{(1)}(T) = (2m+4) - (m-2) = m + 6$.

Equivalently, the corresponding reduced CDAWG grammar can be described as follows. Let $X_{\mathtt{ab}}$ be the nonterminal corresponding to the node (ab), let X_D be the nonterminal corresponding to the node $D = (ab)^m c$, and let X_T be the start nonterminal. After removing the unary rules induced by the in-degree-one nodes $(ab)^2, \ldots, (ab)^{m-1}$, the grammar has the productions

$$X_{ab} \to ab, \qquad X_D \to \underbrace{X_{ab} \cdots X_{ab}}_{m \text{ copies}} c, \qquad X_T \to X_D \, X_D \, \$.$$

The total length of the right-hand sides is $2 + (m+1) + 3 = m + 6$, in agreement with the formula above.

We next compute $\mathsf{g}(T')$. The maximal substrings of T', except for the sink T', are precisely $\varepsilon, (ab), (ab)^2, \ldots, (ab)^m, b, b(ab), b(ab)^2, \ldots, b(ab)^{m-1}$.

The substitution of the first c by b creates the unique factor bb between the two alternating blocks. This makes the strings $b(ab)^r$ left- and right-maximal for $0 \leq r \leq m-1$, while the strings $(ab)^r$ remain left- and right-maximal for $1 \leq r \leq m$. All other factors either have a unique left context or a unique right context, or contain the unique end-marker and hence do not give additional maximal repeats.

Their right-degrees are $\mathsf{d}_{T'}(\varepsilon) = 4$, $\mathsf{d}_{T'}((ab)^m) = 2$, and every other maximal repeat among the above $2m-1$ strings has three right-extensions. Thus $\mathsf{e}(T') = 4 + 3(2m-1) + 2 = 6m + 3$. Furthermore, all the following $2m$ nodes have in-degree one: $(ab), (ab)^2, \ldots, (ab)^m, b, b(ab), b(ab)^2, \ldots, b(ab)^{m-1}$. Hence $\mathrm{v}^{(1)}(T') = 2m$. Again by Fact 2, $\mathsf{g}(T') = \mathsf{e}(T') - \mathrm{v}^{(1)}(T') = (6m+3) - 2m = 4m+3$. In terms of the reduced CDAWG grammar, the situation is particularly simple: all internal maximal-repeat nodes of $\mathsf{CDAWG}(T')$ have in-degree one, and hence the unary rules corresponding to them are removed. Thus the reduced grammar has only the start production

$$X_{T'} \to \underbrace{a\,b \cdots a\,b}_{m \text{ copies of } ab} \; b \; \underbrace{a\,b \cdots a\,b}_{m \text{ copies of } ab} \; c\,\$.$$

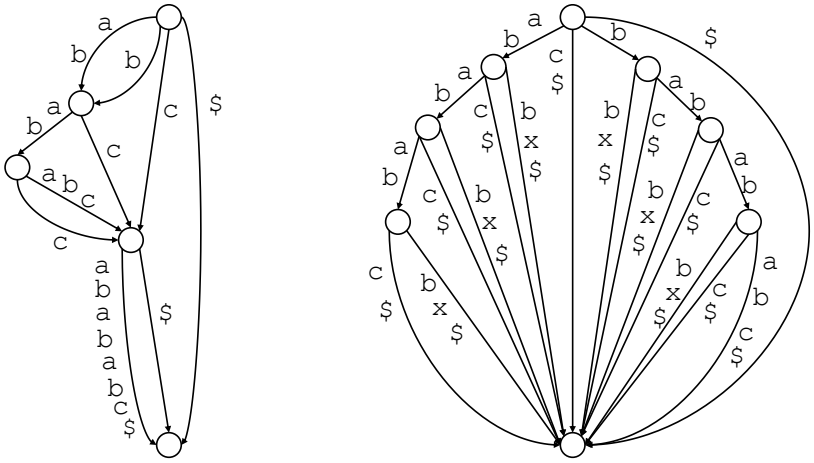

Figure 2. CDAWGs for the lower-bound example with $m = 3$. Here $T = (\mathtt{ab})^3\mathtt{c}(\mathtt{ab})^3\mathtt{c}\$ = \mathtt{abababcababababc}\$$ and $T' = (\mathtt{ab})^3\mathtt{b}(\mathtt{ab})^3\mathtt{c}\$ = \mathtt{abababbababababc}\$ = \mathtt{abababbx}\$$ with $\mathtt{x} = \mathtt{abababc}$. For T, the reduced CDAWG grammar has productions $X_{\mathtt{ab}} \to \mathtt{ab}$, $X_{\mathtt{x}} \to X_{\mathtt{ab}}X_{\mathtt{ab}}X_{\mathtt{ab}}\mathtt{c}$, and $X_T \to X_{\mathtt{x}}X_{\mathtt{x}}\$$. For T', all internal maximal-repeat nodes have in-degree one, and the reduced CDAWG grammar consists of $X_{T'} \to \mathtt{abababbababababc}\$$. We have $\mathsf{e}(T) = 10$, $\mathrm{v}^{(1)}(T) = 1$, and $\mathsf{g}(T) = 9$, while $\mathsf{e}(T') = 21$, $\mathrm{v}^{(1)}(T') = 6$, and $\mathsf{g}(T') = 15$. Hence $\mathsf{g}(T') - \mathsf{g}(T) = 6 = 3\mathsf{g}(T) - 21$.

The length of this right-hand side is $2m + 1 + 2m + 2 = 4m + 3$, again agreeing with $\mathsf{g}(T')$.

Consequently, $\mathsf{g}(T') - \mathsf{g}(T) = (4m + 3) - (m + 6) = 3m - 3$. Since $\mathsf{g}(T) = m + 6$, we obtain $\mathsf{g}(T') - \mathsf{g}(T) = 3\mathsf{g}(T) - 21$. This proves the theorem.

Fig. 2 shows the CDAWGs of the strings T and T' with $m = 3$.

5 Edge-sensitivity ingredients for CDAWG-grammars

In this section, we revisit the CDAWG sensitivity analysis of Hamai et al. [9] and derive an upper bound on $\mathsf{e}(T') - \mathsf{e}(T)$ tailored to our analysis of CDAWG-grammars.

Throughout this paper, T denotes an arbitrarily fixed string and T' denotes the string obtained by performing a single-character edit operation on T. For an insertion or substitution, i denotes the position in T' occupied by the inserted or substituted character. For a deletion of $T[i]$, i denotes the position immediately to the right of the resulting gap in T'.

Definition 2. *Let $x = T'[j..k]$ be a non-empty substring occurrence of T', where $1 \le j \le k \le |T'|$. This occurrence $[j..k]$ of x in T' is called* crossing *if it contains or touches the edit position i. More precisely, it is crossing if $k = i - 1$ or $j \le i \le k$ or $j = i + 1$ for an insertion or a substitution, and if $j \le i \le k + 1$ for a deletion.*

Whenever x has a crossing occurrence, let x_L (resp. x_R) denote its leftmost (resp. rightmost) crossing occurrence in T'. Write $[j_L..k_L]$(resp. $[j_R..k_R]$) for their intervals, where $k_L = j_L + |x| - 1$ and $k_R = j_R + |x| - 1$. We use $S_{x_L} = T'[i..k_L]$ and $S_{x_R} = T'[i..k_R]$ (Fig. 3); with the convention $T'[p..p-1] = \varepsilon$, either string may be empty. Thus these symbols are used only for strings having a crossing occurrence, and their index i is the edit position defined above.

We divide $\mathsf{MR}(T')$ into the following four subsets:

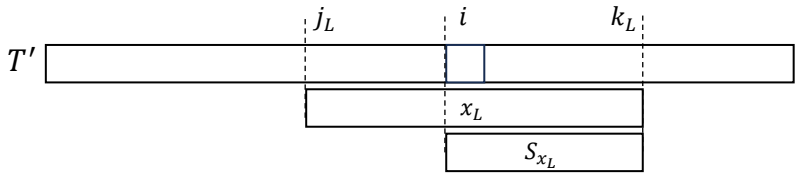

Figure 3. Illustration of x_L in T' for the case where x_L contains i.

Definition 3. *Let* $\mathsf{N} = \mathsf{MR}(T') \setminus \mathsf{MR}(T)$ *and* $\mathsf{Q} = \mathsf{MR}(T') \setminus \mathsf{N}$ *(or equivalently* $\mathsf{Q} = \mathsf{MR}(T') \cap \mathsf{MR}(T)$*). We divide* N *into* N_1*,* N_2*, and* N_3 *as* $\mathsf{N}_1 = \mathsf{N} \cap \mathsf{RightM}(T)$*,* $\mathsf{N}_2 = \mathsf{N} \cap \mathsf{LeftM}(T)$*, and* $\mathsf{N}_3 = \mathsf{N} \backslash (\mathsf{N}_1 \cup \mathsf{N}_2)$*. Further, let* $\mathsf{N}_{3\mathrm{B}}$ *denote the set of strings* $x \in \mathsf{N}_3$ *satisfying the following two conditions: (1) The crossing occurrences of* x *in* T' *have at least two distinct left contexts and at least two distinct right contexts, and (2) There is no distinct right-extension of* x *in* T' *other than the right-extension(s) of the crossing occurrence(s) of* x*. Let* $\mathsf{N}_{3\mathrm{A}} = \mathsf{N}_3 \setminus \mathsf{N}_{3\mathrm{B}}$*.*

Also, we define the number of remaining edges after edits by $\mathsf{rem_e}(T, T') = |\mathsf{E}_T \cap \mathsf{E}_{T'}|$. Thus, $\mathsf{rem_e}(T, T')$ is the number of CDAWG edges of T that remain unchanged after the edit. We write it simply as $\mathsf{rem_e}$ when it is clear from the context.

Upper bound for total out-degrees of nodes w.r.t. $\mathbf{N_1 \cup N_{3A}}$. Here, we show an upper bound for the total out-degrees of nodes corresponding to the strings that are elements of $\mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$.

Lemma 4 (Lemma 2 and 3 of [9]). *If* $x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$*, then there do not exist distinct strings* $y, z \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$ *with* $|x| < |y|$ *and* $|x| < |z|$ *such that both* $S_{x_L} = S_{y_F}$ *and* $S_{x_R} = S_{z_G}$ *hold at the same time, where* $F, G \in \{L, R\}$*.*

We next define, through the map U, the correspondence from $\mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$ to $\mathsf{MR}(T)$. For any $x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$, if there is no $y \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$ ($|y| > |x|, G \in \{L, R\}$) s.t. $S_{x_L} = S_{y_G}$, then we associate x with S_{x_L}, and otherwise we associate x with S_{x_R}.

Let $U(x)$ be the string that is obtained by extending S_{x_G}, to which x corresponds, to the left in T until becoming left-maximal in T. The following holds:

Lemma 5 (Lemmas 4 and 5 of [9]). *If* $x, y \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}$ *and* $x \neq y$*, then* $U(x), U(y) \in \mathsf{M}(T)$ *and* $U(x) \neq U(y)$*. Also,* $\mathsf{d}_{T'}(x) \leq \mathsf{d}_T(U(x)) + 2$*.*

Also, for each $x \in N_1 \cup N_{3A}$, we define $\delta_{\mathsf{N}}(x) = 2 - \max(d_{T'}(x) - d_T(U(x)), 0)$. Since the map U is injective by Lemma 5, $\sum_x \mathsf{d}_T(U(x)) \leq \mathsf{e}(T)$. Then, we have the following: $\sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} \mathsf{d}_{T'}(x) \leq \sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} (\mathsf{d}_T(U(x)) + 2 - \delta_{\mathsf{N}}(x)) \leq \mathsf{e}(T) + 2|\mathsf{M}(T)| - \sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} (\delta_{\mathsf{N}}(x))$.

Lemma 6. $\sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} \mathsf{d}_{T'}(x) \leq \mathsf{e}(T) + 2|\mathsf{M}(T)| - \sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} \delta_{\mathsf{N}}(x)$.

Upper bound for total out-degrees of nodes w.r.t. $\mathbf{N_2 \cup Q}$. Here, we consider an upper bound for the total out-degrees of nodes corresponding to the strings that are elements of $\mathsf{N}_2 \cup \mathsf{Q}$. We divide the total out-edges of the nodes into two: the number $\mathcal{J}(T)$ of *the right-extensions of the non-crossing occurrences* and the number $\mathcal{L}(T)$ of *the right-extensions only in the crossing occurrences*, such that $\sum_{x \in \mathsf{N}_2 \cup \mathsf{Q}} \mathsf{d}_{T'}(x) = \mathcal{J}(T) + \mathcal{L}(T)$.

For $\mathcal{J}(T)$, the right-extensions of x with non-crossing occurrences in T' does not exceed the right-extensions of x in T because the non-crossing occurrences are not affected by the edit. If $x \in \mathsf{N}_2$, then x is not right-maximal in T. Thus, each node has at most one right-extension with non-crossing occurrences. On the other hand, each remaining right-extension of $x \in \mathsf{Q}$ corresponds by definition to an edge in $\mathsf{E}_T \cap \mathsf{E}_{T'}$. By summing up these, we get:

Lemma 7. $\mathcal{J}(T) \leq |\mathsf{N}_2| + \mathsf{rem_e}$.

From now on we consider the second term $\mathcal{L}(T)$. A right-extension $a \in \Sigma$ of $x \in \mathsf{N}_2 \cup \mathsf{Q}$ is said to be a *new* right-extension of x in T', if xa does not occur in T and x has a crossing occurrence that immediately precedes a in T'.

Lemma 8 (Adapted from [9]). *There is an injection from the new right-extensions of all strings in* $\mathsf{N}_2 \cup \mathsf{Q}$ *to the maximal repeats in* $\mathsf{M}(T)$ *(namely to the existing nodes in* $\mathsf{CDAWG}(T)$*), implying* $\mathcal{L}(T) \leq |\mathsf{M}(T)|$.

Putting these results together, the following lemma holds:

Lemma 9. $\sum_{x \in \mathsf{N}_2 \cup \mathsf{Q}} \mathsf{d}_{T'}(x) \leq \mathsf{rem_e} + |\mathsf{M}(T)| + |\mathsf{N}_2|$.

The following upper bound is known for $\mathsf{N}_{3\mathrm{B}}$:

Lemma 10 (Reformulation of Lemma 15 of [9]). $\sum_{x \in \mathsf{N}_{3\mathrm{B}}} \mathsf{d}_{T'}(x) \leq 2|\mathsf{MR}(T)|$.

Recall $|\mathsf{MR}(T)| = |\mathsf{M}(T) \setminus \{T\}| = |\mathsf{M}(T)| - 1$. By Lemmas 6, 9, and 10 we get:

Theorem 11. $\mathsf{e}(T') - \mathsf{e}(T) \leq \mathsf{rem_e} + 5|\mathsf{MR}(T)| - \sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} (\delta_{\mathsf{N}}(x)) + |\mathsf{N}_2| + 3$.

6 Tighter bound for $\mathsf{AS}(\mathsf{g}, n)$

This section analyzes the second half $\mathrm{v}^{(1)}(T) - \mathrm{v}^{(1)}(T')$ of Equation 1 between the original string T and the edited string T'. By combining it with the result from Section 5, we will obtain a tighter bound $\mathsf{AS}(\mathsf{g}, n) \leq 8\mathsf{g} + 4$ (Corollary 27).

Difference in the number of nodes of in-degree one. We consider the number of $\mathsf{C}(T)$ *existing nodes whose in-degrees are changed from one after the edit.* Clearly $\mathrm{v}^{(1)}(T) - \mathrm{v}^{(1)}(T') \leq \mathsf{C}(T)$ holds.

Let $x \in \mathsf{MR}(T)$ be a node of in-degree one in the CDAWG for the original string T before the edit. We consider the following two cases:

Case 1: x is not a maximal repeat in T' after the edit (the node corresponding to x is deleted in $\mathsf{CDAWG}(T')$).
Case 2: x has two or more in-edges in $\mathsf{CDAWG}(T')$ after the edit.

Let V_1 and V_2 denote the sets of strings x in Cases 1 and 2, respectively.

In Case 2, we further define y as the (only) parent of a node $x \in \mathsf{V}_2$ in the CDAWG of the original string T. Also, let q be the longest proper suffix of x that is a maximal repeat in T. See Fig. 4. We categorize strings $x \in \mathsf{V}_2$ using the unique parent y and its suffix q, as follows:

Case 2(a): $y \notin \mathsf{MR}(T')$ (the node for y is deleted in $\mathsf{CDAWG}(T')$).
Case 2(b): $y \in \mathsf{MR}(T')$ and $q \notin \mathsf{MR}(T')$ (the node for q is deleted in $\mathsf{CDAWG}(T')$).

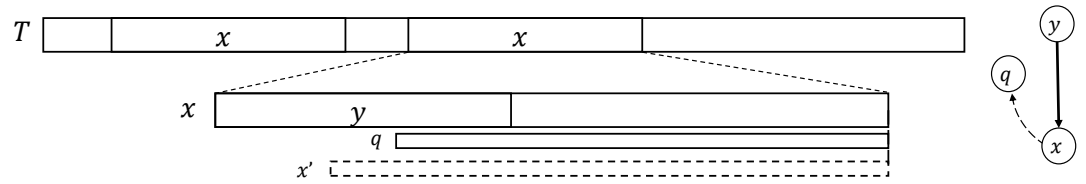

Figure 4. Illustration for the strings y and q for a string $x \in \mathsf{V}_2(T)$ and the relation of their nodes, where the dashed arc represents the suffix link. Any proper suffix x' of x that is longer than q is not a maximal repeat of T.

Case 2(c): $y, q \in \mathsf{MR}(T')$ (both of the nodes for y and q exist in $\mathsf{CDAWG}(T')$).

Let $\mathsf{V}_{2(\mathrm{a})}$, $\mathsf{V}_{2(\mathrm{b})}$, and $\mathsf{V}_{2(\mathrm{c})}$ denote the sets of strings $x \in \mathsf{MR}(T)$ which are in Case 2(a), Case 2(b), and Case 2(c), respectively.

Cases 1, 2(a), 2(b): For each $x \in \mathsf{V}_1 \cup \mathsf{V}_{2(\mathrm{a})} \cup \mathsf{V}_{2(\mathrm{b})}$, we define $f(x) \in \mathsf{MR}(T)$ as follows: (1) $f(x) = x$ if $x \in \mathsf{V}_1$, (2) $f(x) = y$ if $x \in \mathsf{V}_{2(\mathrm{a})}$, and (3) $f(x) = q$ if $x \in \mathsf{V}_{2(\mathrm{b})}$. Also, let $f(V) = \{f(x) \mid x \in V\}$ for each $V \in \{\mathsf{V}_1, \mathsf{V}_{2(\mathrm{a})}, \mathsf{V}_{2(\mathrm{b})}\}$.

By definition, the node for $f(x)$ and its out-edges are deleted in $\mathsf{CDAWG}(T')$. Since we have assumed that T terminates with a unique end-marker \$, $\mathsf{d}_T(f(x)) \geq 2$. Thus, at least $2|f(\mathsf{V}_1) \cup f(\mathsf{V}_{2(\mathrm{a})}) \cup f(\mathsf{V}_{2(\mathrm{b})})|$ edges are removed.

By the definition of $\mathsf{rem_e}$, we have that:

Lemma 12. *At least* $|\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}|$ *existing edges are removed after edits. Then,* $\mathsf{e}(T) \geq \mathsf{rem_e} + (|\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}|)$.

For $f(x)$, we have the following lemmas.

Lemma 13. *If* $x_1, x_2 \in \mathsf{V}_{2(\mathrm{a})}$ *and* $x_1 \neq x_2$*, then* $y_1 \neq y_2$.

Proof. Suppose that $y_1 = y_2$. Then y_1 is the longest proper prefix of both x_1 and x_2 that is a maximal repeat in T, which implies that $a \neq b$ where $a, b \in \Sigma$, $y_1 a$ is a prefix of x_1, and $y_1 b$ is a prefix of x_2. Since x_1, x_2 are substrings of T' after the edit, $y_1 \in \mathsf{M}(T')$, which is a contradiction. □

By symmetry, the following lemma also holds.

Lemma 14. *If* $x_1, x_2 \in \mathsf{V}_{2(\mathrm{b})}$ *and* $x_1 \neq x_2$*, then* $q_1 \neq q_2$.

Lemma 15. *If* $x_1 \in \mathsf{V}_{2(\mathrm{a})}$ *and* $x_2 \in \mathsf{V}_{2(\mathrm{b})}$*, then* $f(x_1) \neq f(x_2)$.

Proof. Suppose that $f(x_1) = f(x_2)$. By definition, $f(x_1)$ is a prefix of x_1 and a suffix of x_2. Since $x_1, x_2 \in \mathsf{M}(T')$, $f(x_1) \in \mathsf{M}(T')$, which is a contradiction. □

By Lemmas 13 and 14, there is an injective mapping between V and $f(V)$ for each $V \in \{\mathsf{V}_1, \mathsf{V}_{2(\mathrm{a})}, \mathsf{V}_{2(\mathrm{b})}\}$. Thus, $|V| = |f(V)|$ holds for each V. Also, for each element x, x can be counted twice in $f(\mathsf{V}_1)$ and $f(\mathsf{V}_{2(\mathrm{a})})$ or $f(\mathsf{V}_{2(\mathrm{b})})$, while there is no $x \in f(\mathsf{V}_{2(\mathrm{a})}) \cap f(\mathsf{V}_{2(\mathrm{b})})$ by Lemma 15.

As a result, $|\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}| \leq 2|f(\mathsf{V}_1) \cup f(\mathsf{V}_{2(\mathrm{a})}) \cup f(\mathsf{V}_{2(\mathrm{b})})|$ holds. The number of removed edges is at least the quantity on the right-hand side, which implies that at least $|\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}|$ edges are removed.

Case 2(c): We now consider the case $x \in \mathsf{V}_{2(\mathrm{c})}$. If the in-degree of x is at least 2 in $\mathsf{CDAWG}(T')$, the following lemma holds: For accessibility, recall that $\mathsf{I}_T(x) = \{y \mid (y, x) \in \mathsf{E}_T\}$ is the set of in-neighbors of x in $\mathsf{CDAWG}(T)$.

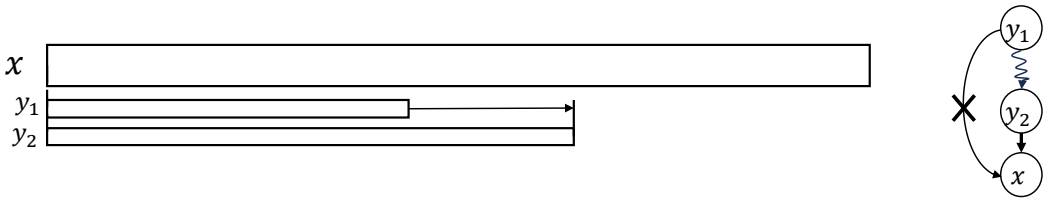

Figure 5. Illustration for the proof of Lemma 16. The case does not occur.

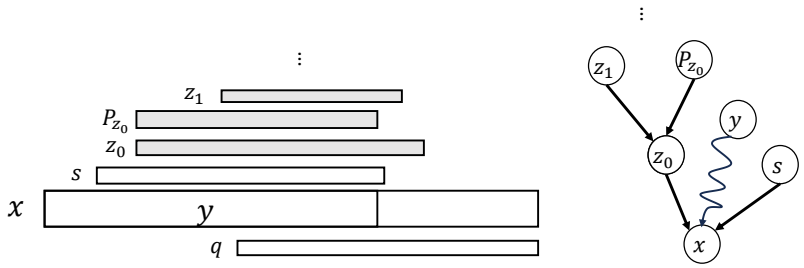

Figure 6. Illustration for z_k and P_{z_k} in x and $\mathsf{CDAWG}(T')$. Both s and z_0 are elements of $I_{T'}(x)$. Among them, z_0 is chosen so that its occurrence ends at the rightmost position.

Lemma 16. *If $y_1, y_2 \in \mathsf{I}_{T'}(x)$, at least one of y_1, y_2 has a non-prefix occurrence in x.*

Proof. Assume, to the contrary, that each of y_1 and y_2 occurs exactly once in x, as a prefix of x (see Fig. 5). Without loss of generality, let $|y_1| < |y_2|$. Let a be the character such that $y_1 a$ is a prefix of x. Since y_1 and y_2 are prefixes of x, the right-representative $\mathsf{rexp}_{T'}(y_1 a)$ of $y_1 a$ is not longer than y_2 and thus it is not x. If there is no other occurrence of y_1 in x, then $(y_1, x) \notin \mathsf{E}_{T'}$, but this is a contradiction. Thus, at least one of y_1, y_2 has a non-prefix occurrence in x. □

By Lemma 16, if $\mathsf{I}_{T'}(x)$ has at least two elements, then at least one in-neighbor in $\mathsf{I}_{T'}(x)$ has a non-prefix occurrence in x. Now, for $k \geq 0$, we define z_k and P_{z_k} as follows:

- z_0 is the maximal repeat in $\mathsf{I}_{T'}(x)$ whose ending position within x, denoted $j^{(x)}_{z_0}$, is the rightmost among those in $\mathsf{I}_{T'}(x)$.
- P_{z_k} is the maximal repeat which is the longest prefix of z_k where $P_{z_k} \in \mathsf{I}_{T'}(z_k)$.
- z_{k+1} is the maximal repeat in $\mathsf{I}_{T'}(z_k)$ whose ending position within z_k in x, denoted $j^{(x)}_{z_{k+1}}$, is the rightmost among those in $\mathsf{I}_{T'}(z_k)$.

If there exist two or more maximal repeats with the same rightmost ending position, then the shortest one is selected as z_k. P_{z_k} and z_{k+1} are undefined when z_k is a suffix of y or z_k is of in-degree one. See also Fig. 6.

For any k, let $i^{(x)}_{z_k}$ be the beginning position of z_k in x. Since $z_{k+1} \in \mathsf{I}_{T'}(z_k)$, we have $(z_{k+1}, z_k) \in \mathsf{E}_{T'}$, $|z_k| > |z_{k+1}|$ holds. Therefore, the recursive construction of P_{z_k} and z_{k+1} stops at some z_k.

Let $i^{(x)}_y$ and $j^{(x)}_y$ (resp. $i^{(x)}_q$ and $j^{(x)}_q$) denote the beginning and ending positions of y (resp. q) as prefix (resp. suffix) of x. For q and z_k, we have:

Lemma 17. *For $k \geq 0$, $i^{(x)}_{z_k} < i^{(x)}_q < j^{(x)}_y \leq j^{(x)}_{z_k}$ holds.*

Proof. We prove the lemma by induction (see also Fig. 7).

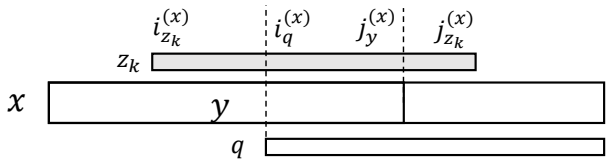

Figure 7. Illustration for Lemma 17.

Let $z'_0 = x[j^{(x)}_{z_0} + 1..|x|]$. Then $z_0 z'_0$ is a suffix of x, and z'_0 is the label of the edge (z_0, x) in $\mathsf{CDAWG}(T')$. Assume on the contrary that $i^{(x)}_{z_0} \geq i^{(x)}_q$. Then $z_0 z'_0$ is also a suffix of q and z'_0 is the label of the edge (z_0, q), which is a contradiction. Assume on the contrary that $j^{(x)}_y > j^{(x)}_{z_0}$. Then z_0 is a substring of y. Since $y \in \mathsf{MR}(T')$, every path from z_0 to x goes through the node y. This implies that z_0 does not have an out-edge to x, a contradiction. By the definition of q, any suffix of x that is longer than q is represented by the same node as x. Since x has in-degree one in $\mathsf{CDAWG}(T)$ and its unique in-neighbor is y, y and q must overlap in x. Hence $i^{(x)}_q < j^{(x)}_y$. Thus, $i^{(x)}_{z_0} < i^{(x)}_q < j^{(x)}_y \leq j^{(x)}_{z_0}$.

If $i^{(x)}_{z_k} < i^{(x)}_q < j^{(x)}_y \leq j^{(x)}_{z_k}$ holds and z_{k+1} is generated, then $j^{(x)}_y < j^{(x)}_{z_k}$ holds by definition. Assume on the contrary that $i^{(x)}_{z_{k+1}} \geq i^{(x)}_q$. Let s_{z_k} denote the overlap between z_k and q. Then z_{k+1} is a substring of s_{z_k}. Since $z_k, q \in \mathsf{MR}(T')$, $s_{z_k} \in \mathsf{MR}(T')$. Thus, every path from z_{k+1} to z_k goes through the node s_{z_k} (and thus z_{k+1} does not have an out-edge to z_k), a contradiction. Assume on the contrary that $j^{(x)}_y > j^{(x)}_{z_{k+1}}$. Then z_{k+1} is a substring of y. Since $y \in \mathsf{MR}(T')$, every path from z_{k+1} to x goes through the node y. This implies that z_{k+1} does not have an out-edge to z_k, which is a contradiction. Thus, $i^{(x)}_{z_{k+1}} < i^{(x)}_q < j^{(x)}_y \leq j^{(x)}_{z_{k+1}}$. □

Consider the string z_k such that z_{k+1} is not generated. Let $z_l(x) = z_k$. In the sequel, we consider two sub-cases of Case 2(c):

Case 2(c)-1: $z_l(x)$ has only a single in-edge and it is not a suffix of y. Let $\mathsf{V}_{2(\mathrm{c})-1}(T)$ denote the set of strings x in Case 2(c)-1. We remark that $z_l(x)$ has only a single in-edge, and $z_l(x)$ is a new node whose in-degree is one in $\mathsf{CDAWG}(T')$, instead of x. Also, we have the following lemma:

Lemma 18. *If x_1 and x_2 are distinct strings in Case 2(c)-1, then $z_l(x_1) \neq z_l(x_2)$.*

Proof. Suppose that $z_l(x_1) = z_l(x_2)$. Let x' be the maximal common substring of x_1 and x_2 which contains $z_l(x_1) = z_l(x_2)$. By the maximality of x' as a common substring of the two occurrences in T, x' is a maximal repeat in T. Also, let y_1 (resp. q_1) denote the longest proper prefix (resp. suffix) of x_1 that is in $\mathsf{MR}(T)$. Since $i^{(x_1)}_{z_l(x_1)} < i^{(x_1)}_{q_1} < j^{(x_1)}_{y_1} \leq j^{(x_1)}_{z_l(x_1)}$ by Lemma 17 and $z_l(x_1)$ is not a suffix of y_1, $i^{(x_1)}_{x'} < i^{(x_1)}_{q_1} < j^{(x_1)}_{y_1} < j^{(x_1)}_{x'}$ also holds. This also implies that $x' \neq y_1$. Since y_1 is the only node that has an out-edge to x_1 in $\mathsf{CDAWG}(T)$, and q_1 is the longest maximal repeat of T that is a proper suffix of x_1, we derive a contradiction as follows:

If the considered occurrence of x' is a suffix of x_1, then x' itself is a suffix of x_1 longer than q_1, contradicting the definition of q_1. Otherwise, let $a \in \Sigma$ be the character immediately following this occurrence of x' in x_1, so that $x'a \in \mathsf{Substr}(x_1)$. Then $\mathsf{rexp}_T(x'a)$ is a suffix of x_1 that is longer than q_1. If $\mathsf{rexp}_T(x'a) \neq x_1$, then this contradicts the definition of q_1. Otherwise, $\mathsf{rexp}_T(x'a) = x_1$, and hence x' has an outgoing edge to x_1 in $\mathsf{CDAWG}(T)$, contradicting that y_1 is the unique in-neighbor of x_1 in $\mathsf{CDAWG}(T)$. □

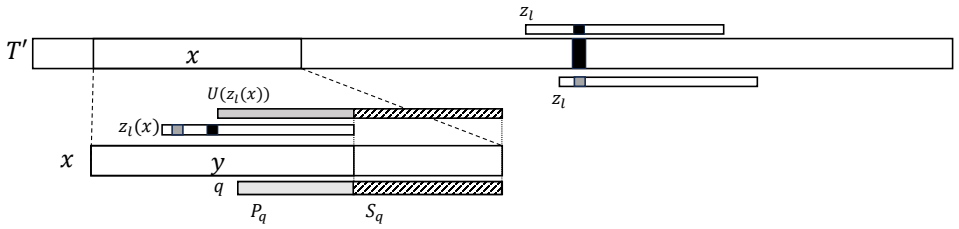

Figure 8. Illustration for the proof of Lemma 19. The black position in T' is the edited position. P_q has a non-crossing occurrence which does not contain nor touch the edited position.

By Lemma 18, there is an injection that maps a new node $z_l(x)$ in $\mathsf{CDAWG}(T')$ to an existing node x in $\mathsf{CDAWG}(T)$, and these nodes have only a single in-edge. Thus, the number of nodes whose in-degree is one does not change in Case 2(c)-1.

Case 2(c)-2: $z_l(x)$ is a suffix of y. Let $\mathsf{V}_{2(\mathrm{c})-2}$ be the set of strings x in Case 2(c)-2, and $\mathsf{V}^{(y)}_{2(\mathrm{c})-2}$ be the set of strings y corresponding to $x \in \mathsf{V}_{2(\mathrm{c})-2}$. If $x \in \mathsf{V}_{2(\mathrm{c})-2}$, $z_l(x)$ is right-maximal and is not left-maximal in T since $z_l(x)$ is a suffix of y and $z_l(x) \notin \mathsf{MR}(T)$. Then, $z_l(x) \in \mathsf{N}_1$. Let $P_q(x)$ denote the overlap of y and q. Since $y, q \in \mathsf{MR}(T)$, $P_q \in \mathsf{MR}(T)$. For $P_q(x)$, we have:

Lemma 19. *For every $x \in \mathsf{V}_{2(\mathrm{c})-2}$, $P_q = P_q(x)$ also has a crossing-occurrence in T'. Furthermore, $U(z_l(x))$ (the existing maximal repeat corresponding to $z_l(x)$) is a suffix of P_q.*

Proof. Suppose P_q has only non-crossing occurrence(s) in T'. See Fig. 8. $z_l(x)$ has a crossing occurrence in T' since $z_l(x) \in \mathsf{N}_1$. Since $z_l(x)$ has a crossing occurrence whereas P_q is assumed to have none and both $U(z_l(x))$ and P_q are suffixes of $z_l(x)$, $|U(z_l(x))| > |P_q|$ (also see Fig. 8). Thus $U(z_l(x))$ is a maximal repeat of T having P_q as a proper suffix. Let S_q be the string such that $q = P_q S_q$. Since $U(z_l(x))S_q$ is a suffix of x longer than q, $U(z_l(x))S_q \equiv^{\mathrm{L}}_T x$ holds. Since $U(z_l(x))$ is also a suffix of y, there exists a path from $U(z_l(x))$ to x which does not contain y in $\mathsf{CDAWG}(T)$. This implies that x has another in-edge from some node other than y in $\mathsf{CDAWG}(T)$ before edit, which is a contradiction. By the above arguments, if $U(z_l(x))$ is longer than P_q, then it also leads to a contradiction. Since $U(z_l(x))$ and P_q are suffixes of y, $U(z_l(x))$ is also a suffix of P_q. □

For the correspondence between y and $z_l(x), P_q$, we have the following lemma:

Lemma 20. *Let $y_1, y_2 \in V^{(y)}_{2(c)-2}$ be distinct. For any $x_1, x_2 \in V_{2(c)-2}$ corresponding to y_1, y_2, respectively, we have $z_l(x_1) \neq z_l(x_2)$ and $P_q(x_1) \neq P_q(x_2)$.*

Proof. Assume $z_l(x_1) = z_l(x_2)$. Since $z_l(x_1)$ occurs both in y_1 and in y_2 as their suffix, there exists a maximal repeat of T in $z_l(x_1)$ or in a suffix of y_1 that is longer than $z_l(x_1)$. This implies that there exists a maximal repeat of T that is longer than $P_q(x_1)$ (similar to Lemma 19), which is a contradiction. Thus, $z_l(x_1) \neq z_l(x_2)$. Assume $P_q(x_1) = P_q(x_2)$. Since this string occurs as a suffix of both y_1 and y_2, the same argument as above implies that there is a maximal repeat of T contained in $P_q(x_1)$ or in a suffix of y_1 that is longer than $P_q(x_1)$, which is a contradiction. □

By Lemma 20, there exists an injective mapping from y to $z_l(x)$ and P_q.

We further show the value of $|\mathsf{V}^{(y)}_{2(\mathrm{c})-2}|$ by using $z_l(x)$. Since $z_l(x)$ is a suffix of $y \in \mathsf{MR}(T)$, $z_l(x) \in \mathsf{N}_1$, which implies $\mathsf{d}_{T'}(z_l(x)) \leq \mathsf{d}_T(U(z_l(x))) + 2$ due to Lemma 5.

Let $\mathsf{V}^{(z)}_{2(\mathrm{c})-2}$ denote the set of $z_l(x)$ where $x \in \mathsf{V}_{2(\mathrm{c})-2}$. Let $\delta_{\mathsf{V}}(z_l(x))$ denote the value such that $\delta_{\mathsf{V}}(z_l(x)) = \max(\mathsf{d}_{T'}(z_l(x)) - \mathsf{d}_T(U(z_l(x))), 0)$, and x_y denote the arbitrarily chosen x corresponding to each $y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}$. Also, the following lemma holds:

Lemma 21. $|\mathsf{N}_2| \leq |\mathsf{M}(T)| - \sum_{y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} \delta_{\mathsf{V}}(z_l(x_y))$.

Proof. For each $y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}$, let $q = q(x_y)$ and $P_q = P_q(x_y)$. Since $q, y \in Q$ and P_q is their overlap in the occurrence inside x_y, we have that P_q is left-maximal and right-maximal in both T and T'. Hence, $P_q \in \mathsf{Q}$. Also, by Lemma 19, $U(z_l(x_y))$ is a suffix of $P_q(x_y)$, and by Lemma 17 and the definition of Case 2(c)-2, $P_q(x_y)$ is a suffix of $z_l(x_y)$.

Suppose $\delta_{\mathsf{V}}(z_l(x_y)) = 2$. Since $U(z_l(x_y))$ is a suffix of $P_q(x_y)$ and $P_q(x_y)$ is a suffix of $z_l(x_y)$, $\mathsf{d}_{T'}(P_q(x_y)) \geq \mathsf{d}_T(U(z_l(x_y))) + 2 \geq \mathsf{d}_T(P_q(x_y)) + 2$ holds. This implies that $P_q(x_y) \in Q$ has at least two new right-extensions in T'. By Lemma 8, these new right-extensions are injectively mapped to two distinct maximal repeats in $\mathsf{M}(T)$. Similarly, for each $z_l(x_y)$ where $\delta_{\mathsf{V}}(z_l(x_y)) = 1$, $\mathsf{d}_{T'}(P_q(x_y)) \geq \mathsf{d}_T(P_q(x_y)) + 1$ holds, and $P_q(x_y) \in Q$ has at least one new right-extension in T', which is injectively mapped to a maximal repeat in $\mathsf{M}(T)$ by Lemma 8. By Lemma 20, the strings $P_q(x_y)$ for distinct $y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}$ are pairwise distinct. Then, at least $\sum_{y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} \delta_{\mathsf{V}}(z_l(x_y))$ new right-extensions of these strings $P_q(x_y) \in \mathsf{Q}$ are injectively mapped to distinct maximal repeats in $\mathsf{M}(T)$. On the other hand, each string $s \in \mathsf{N}_2$ has at least one new right-extension in T', since $s \in \mathsf{M}(T')$ but $s \notin \mathsf{RightM}(T)$. By Lemma 8, these new right-extensions are also injectively mapped to distinct maximal repeats in $\mathsf{M}(T)$. Applying the injection of Lemma 8 to the union of the above new right-extensions of strings in $\mathsf{N}_2 \cup \mathsf{Q}$, we obtain $|\mathsf{N}_2| + \sum_{y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} \delta_{\mathsf{V}}(z_l(x_y)) \leq |\mathsf{M}(T)|$. □

Since $z_l(x) \in \mathsf{N}_1$ and $\delta_{\mathsf{N}}(z_l(x)) = 2 - \max(\mathsf{d}_{T'}(z_l(x)) - \mathsf{d}_T(U(z_l(x))), 0)$, we obtain:

Corollary 22. *For every* $x \in \mathsf{V}_{2(\mathrm{c})-2}$, $\delta_{\mathsf{N}}(z_l(x)) + \delta_{\mathsf{V}}(z_l(x)) = 2$.

By Lemma 20, the chosen strings $z_l(x_y)$ are pairwise distinct elements of $\mathsf{V}^{(z)}_{2(\mathrm{c})-2}$. Hence:

Corollary 23. $|\mathsf{N}_1| \geq |\mathsf{V}^{(z)}_{2(\mathrm{c})-2}| \geq |\mathsf{V}^{(y)}_{2(\mathrm{c})-2}|$ *holds.*

To consider the size of $|\mathsf{V}_{2(\mathrm{c})-2}|$, let $D = \sum_{y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} (\mathsf{d}_T(y))$, and X be the total number of in-edges of $x \in \mathsf{V}_{2(\mathrm{c})-2}$ in $\mathsf{CDAWG}(T)$. See the upper diagram (a) of Fig. 9. Since $x \in \mathsf{V}_{2(\mathrm{c})-2}$ has only one in-edge (y, x) where $y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}$ in $\mathsf{CDAWG}(T)$, $|\mathsf{V}_{2(\mathrm{c})-2}| = X \leq D$ holds.

We now leave the analysis of Case 2(c)-2 and combine all cases globally. Recall that $\mathsf{C}(T)$ counts the existing CDAWG nodes whose in-degree changes from one after the edit. Putting Cases 1, 2(a), 2(b), 2(c)-1, and 2(c)-2 together gives the following bound. Since $\mathsf{C}(T) = |\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}| + |\mathsf{V}_{2(\mathrm{c})-1}(T)| + |\mathsf{V}_{2(\mathrm{c})-2}|$ and at least $|\mathsf{V}_{2(\mathrm{c})-1}(T)|$ new nodes have only one in-edge in Case 2(c)-1, we have the following:

Lemma 24. $\mathrm{v}^{(1)}(T) - \mathrm{v}^{(1)}(T') \leq |\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}| + X$ *holds.*

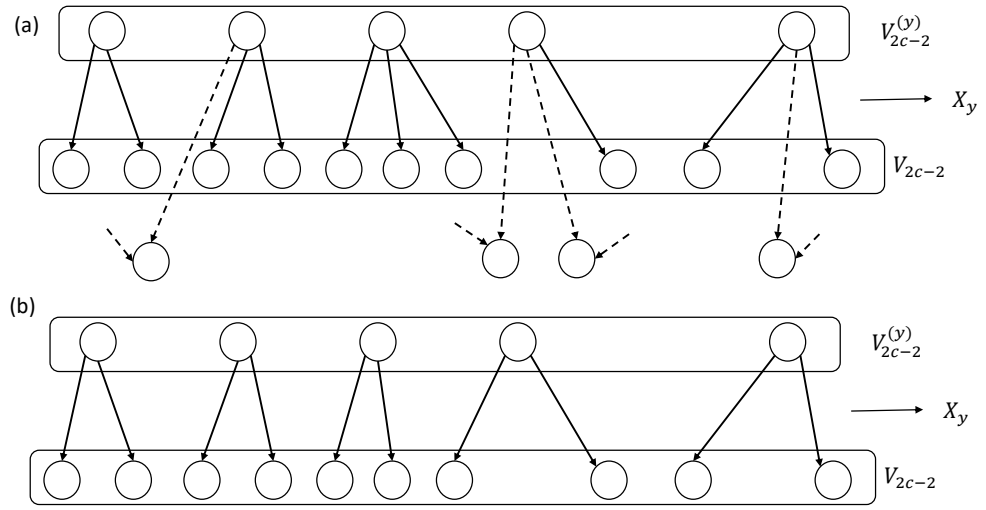

Figure 9. (a) Illustration for the definitions of $\mathsf{V}^{(y)}_{2(\mathrm{c})-2}$, X and D. D counts all of the out-edges from $y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}$, although X only counts the edges going to $x \in \mathsf{V}_{2(\mathrm{c})-2}$ (only solid-line edges are counted). (b) The situation of (a) in which $\mathsf{g}(T') - \mathsf{g}(T)$ is maximized.

Finally, we combine the results of Theorem 11, Lemmas 12, 21 and 24, and Corollaries 22 and 23. Then, we have:

$$\begin{aligned}
&\mathsf{g}(T') - \mathsf{g}(T) \\
&= (\mathsf{e}(T') - \mathsf{e}(T)) + (\mathrm{v}^{(1)}(T) - \mathrm{v}^{(1)}(T')) \\
&\leq \mathsf{rem_e} + 5|\mathsf{MR}(T)| - \textstyle\sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} \delta_{\mathsf{N}}(x) + |\mathsf{N}_2| + 3 + |\mathsf{V}_1| + |\mathsf{V}_{2(\mathrm{a})}| + |\mathsf{V}_{2(\mathrm{b})}| + X \\
&\leq \mathsf{e}(T) + 6|\mathsf{MR}(T)| - \textstyle\sum_{x \in \mathsf{N}_1 \cup \mathsf{N}_{3\mathrm{A}}} \delta_{\mathsf{N}}(x) - \sum_{y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} \delta_{\mathsf{V}}(z_l(x_y)) + X + 4 \\
&\leq \mathsf{e}(T) + 6|\mathsf{MR}(T)| - \textstyle\sum_{y \in \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} (\delta_{\mathsf{N}}(z_l(x_y)) + \delta_{\mathsf{V}}(z_l(x_y))) + X + 4 \\
&\leq \mathsf{e}(T) + 6|\mathsf{MR}(T)| + X - 2|\mathsf{V}^{(y)}_{2(\mathrm{c})-2}| + 4.
\end{aligned}$$

We call this Inequality (α).

Now we consider the value of X. Let $\hat{D} = \sum_{y \in \mathsf{MR}(T) \setminus \mathsf{V}^{(y)}_{2(\mathrm{c})-2}} (\mathsf{d}_T(y))$. For D, $|\mathsf{MR}(T)|$ and $\mathsf{e}(T)$, the following holds:

Lemma 25. *Let* $\omega = X - 2|\mathsf{V}^{(y)}_{2(\mathrm{c})-2}|$, *then* $|\mathsf{MR}(T)| \leq \min(\mathsf{e}(T), \mathsf{e}(T) - \omega)/2$.

Proof. If $\omega \leq 0$, then $\mathsf{e}(T) \geq 2|\mathsf{MR}(T)|$ due to Fact 1. If $\omega > 0$, then $D \geq X = 2|\mathsf{V}^{(y)}_{2(\mathrm{c})-2}| + \omega$ holds. Since $D + \hat{D} = \mathsf{e}(T)$ and $\hat{D} \geq 2|\mathsf{MR}(T) \setminus \mathsf{V}^{(y)}_{2(\mathrm{c})-2}|$, $\mathsf{e}(T) \geq 2|\mathsf{MR}(T)| + \omega$. Then $|\mathsf{MR}(T)| \leq (\mathsf{e}(T) - \omega)/2$. □

By substituting the value of Lemma 25 to Inequality (α) above, we get $\mathsf{g}(T') - \mathsf{g}(T) \leq 4\mathsf{e}(T) + \omega + 4$ if $\omega \leq 0$, or $\mathsf{g}(T') - \mathsf{g}(T) \leq 4\mathsf{e}(T) - 2\omega + 4$ if $\omega > 0$. Therefore, $\mathsf{g}(T') - \mathsf{g}(T) \leq 4\mathsf{e}(T) + 4$. Fig. 9(b) illustrates the situation suggested by the above bound. As a result, we have the following theorem:

Theorem 26. *The additive sensitivity of the CDAWG-grammar is at most* $4\mathsf{e} + 4$.

Since $\mathsf{g}(T) \geq \mathsf{e}(T)/2$, Theorem 26 immediately leads to the following:

Corollary 27. *The additive sensitivity of the CDAWG-grammar is at most* $8\mathsf{g} + 4$.

References

1. T. Akagi, M. Funakoshi, and S. Inenaga: *Sensitivity of string compressors and repetitiveness measures.* Information and Computation, 291 2023, p. 104999.
2. P. Baturo, M. Piatkowski, and W. Rytter: *Usefulness of directed acyclic subword graphs in problems related to Standard Sturmian Words.* Int. J. Found. Comput. Sci., 20(6) 2009, pp. 1005–1023.
3. D. Belazzougui and F. Cunial: *Fast label extraction in the CDAWG*, in SPIRE 2017, 2017, pp. 161–175.
4. D. Belazzougui and F. Cunial: *Representing the Suffix Tree with the CDAWG*, in 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), vol. 78, 2017, pp. 7:1–7:13.
5. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* Journal of the ACM, 34(3) 1987, pp. 578–595.
6. A. Cleary and J. Dood: *Constructing the CDAWG CFG using LCP-intervals*, in 2023 Data Compression Conference (DCC), 2023, pp. 178–187.
7. M. Crochemore and R. Vérin: *On compact directed acyclic word graphs*, in Structures in Logic and Computer Science: A Selection of Essays in Honor of A. Ehrenfeucht, Springer, 1997, pp. 192–211.
8. H. Fujimaru, Y. Nakashima, and S. Inenaga: *Tight bounds for the sensitivity of CDAWGs with left-end edits.* Acta Informatica, 62(12) 2025.
9. R. Hamai, H. Fujimaru, and S. Inenaga: *Constant sensitivity on the CDAWGs*, in Proc. the 37th Annual Symposium on Combinatorial Pattern Matching (CPM 2026), 2026, To appear.
10. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi: *On-line construction of compact directed acyclic word graphs.* Discrete Applied Mathematics, 146(2) 2005, pp. 156–179.
11. J. Radoszewski and W. Rytter: *On the structure of compacted subword graphs of Thue-Morse words and their applications.* J. Discrete Algorithms, 11 2012, pp. 15–24.
12. W. Rytter: *The structure of subword graphs and suffix trees of Fibonacci words.* Theor. Comput. Sci., 363(2) 2006, pp. 211–223.
13. T. Takagi, K. Goto, Y. Fujishige, S. Inenaga, and H. Arimura: *Linear-size CDAWG: New repetition-aware indexing and grammar compression*, in SPIRE 2017, 2017, pp. 304–316.
14. M. Takeda, T. Matsumoto, T. Fukuda, and I. Nanri: *Discovering characteristic expressions from literary works: a new text analysis method beyond n-gram statistics and KWIC*, in Discovery Science 2000, 2000, pp. 112–126.
15. P. Weiner: *Linear pattern matching algorithms*, in Proceedings of the 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 1–11.

Relaxation of Square-Freeness

Hiroki Shibata[1], Takuya Mieno[2], Dominik Köppl[3], and Shunsuke Inenaga[4]

[1] Joint Graduate School of Mathematics for Innovation, Kyushu University
shibata.hiroki.753@s.kyushu-u.ac.jp
[2] Graduate School of Informatics and Engineering, University of Electro-Communications
tmieno@uec.ac.jp
[3] Department of Computer Science and Engineering, University of Yamanashi
dkppl@yamanashi.ac.jp
[4] Department of Informatics, Kyushu University
inenaga.shunsuke.380@m.kyushu-u.ac.jp

Abstract. We extend the analysis of nonrepetitive sequences of Entringer et al. [Journal of Combinatorial Theory, 1974] to relaxations of equality testing under nonstandard equivalence relations, in particular parameterized equivalence and order-preserving equivalence. For this setting, we introduce ℓ^+-squares, defined as squares whose total length is at least 2ℓ. We obtain an infinite 3^+-parameterized-square-free ternary word and an infinite 3^+-order-preserving-square-free binary word through an approach based on combinatorics on words. In addition, we report the longest ℓ^+-square-free words across several equivalence relations.

Keywords: squarefree words, parameterized equivalence, order-preserving equivalence, morphic word

1 Introduction

A *square* is a non-empty word of the form uu. Squares are one of the simplest and most studied repetitions in words [4,9]. A word is *square-free* if it contains no square as a subword. More than a century ago, Thue [21] constructed an infinite square-free word over a ternary alphabet. A ternary alphabet is the smallest possible alphabet for an infinite square-free word. Indeed, every infinite binary sequence contains at least one of the squares aa, bb, and $abab$.

The notion of a square can be generalized by replacing equality between the two halves with a weaker equivalence relation on words. In this paper, we use two equivalence relations from pattern matching: *parameterized equivalence* [2] and *order-preserving equivalence* [12]. We say that two words u and v of the same length ℓ are *parameterized equivalent* if there is a bijection f with $f(u[i]) = v[i]$ for each text position i. We say that they are *order-preserving equivalent* if $u[i] \preceq u[j] \Leftrightarrow v[i] \preceq v[j]$ holds for every pair of text positions i and j. Order-preserving equivalence can also be viewed through the *increasing bijection* on the alphabet. Equivalently, u and v are parameterized equivalent under an increasing bijection. A non-empty word uv is called a *parameterized* (resp. *order-preserving*) *square* if u and v are parameterized (resp. order-preserving) equivalent [13].

Some equivalence relations admit no infinite square-free words. For example, every word of length two is a parameterized square, so no parameterized-square-free infinite word can exist. Even if we consider only *non-trivial* squares (those of length greater than two), no infinite word avoids every non-trivial parameterized square [13]. Consequently, weaker notions of square-freeness are required to give positive results on infinite words that avoid a weaker type of square. We study ℓ^+*-squares*, squares of

Hiroki Shibata, Takuya Mieno, Dominik Köppl, Shunsuke Inenaga: Relaxation of Square-Freeness, pp. 58–70.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

Table 1. Lengths of the longest ℓ^+-square-free words under various equivalence relations and alphabet sizes. An entry marked ∞ indicates the existence of an infinite ℓ^+-square-free word. A question mark (?) denotes a case that remains open. A dash ($-$) indicates that infiniteness follows from another case (e.g., a smaller ℓ or σ).

Strict Equality

$\sigma\backslash\ell$	1	2	3	4
2	3	18	∞	$-$
3	∞	$-$	$-$	$-$
4	$-$	$-$	$-$	$-$
5	$-$	$-$	$-$	$-$

Parameterized Equivalence

$\sigma\backslash\ell$	1	2	3	4
2	1	7	∞	$-$
3	1	9	$-$	$-$
4	1	9	$-$	$-$
5	1	9	$-$	$-$

Order-Preserving Equivalence

$\sigma\backslash\ell$	1	2	3	4
2	1	7	∞	$-$
3	1	∞	$-$	$-$
4	1	$-$	$-$	$-$
5	1	$-$	$-$	$-$

Cartesian-Tree Equivalence [17]

$\sigma\backslash\ell$	1	2	3	4
2	1	5	29	?
3	1	9	?	?
4	1	9	?	?
5	1	9	?	?

length at least 2ℓ, and investigate ℓ^+*-square-free words* under parameterized equivalence.

A complete characterization of the avoidability of length-restricted parameterized and order-preserving squares can be derived by combining existing results [13,16]. However, the proofs by Ng et al. [16] do not rely solely on combinatorics on words because their approach uses large morphisms and automated theorem proving tools. In this paper, we construct square-free words using an approach based on combinatorics on words.

Our main results are summarized as follows. We first prove the existence of an infinite 3^+-parameterized-square-free ternary word. We then show the existence of an infinite 3^+-order-preserving-square-free binary word. Both infinite words are constructed using small morphisms, and their square-freeness is proved through an approach based on combinatorics on words. Finally, through computational experiments, we establish the maximum lengths of words that avoid length-restricted squares under several nonstandard equivalence relations and give explicit longest words.

Table 1 summarizes the maximum lengths of ℓ^+-square-free words under several equivalence relations.

Related Work. Square-free words were first studied by Thue [21] over a century ago and have since remained an active topic of research. Variants involving length-restricted squares have also been explored. For example, it is known that every binary word of length greater than eighteen contains a 2^+-square [8]. In contrast, infinite binary 3^+-square-free words exist [8,20]; they are constructed by applying a morphism to a ternary square-free word. This kind of freeness has also been studied for Gray codes [18] and circular words [5].

Squares and other repetitions under generalized equivalence relations on words have also been studied. In the abelian model, where equivalence is defined by the multiset of characters, Keränen [11] showed the existence of infinite abelian-square-free words over a four-character alphabet, but it is known that none exist over a ternary alphabet. In the parameterized setting, the number of parameterized squares

in a word has also been studied [13,10]. Cubic and unary patterns under permutations, which are closely related to parameterized matching, have also been studied [14,6].

Words avoiding parameterized and order-preserving squares have also been studied in several works. Kociumaka et al. [13] showed that an infinite 2^+-order-preserving-square-free word exists over a ternary alphabet, whereas no such word exists in the parameterized model even without restrictions on the alphabet size. Ng et al. [16] constructed an infinite binary word that avoids every ordinary square and *antisquare* of length at least 6, where an antisquare is a binary word of the form $x\overline{x}$ and $\overline{x}$ is obtained by changing every character of x. Since parameterized squares over a binary alphabet are precisely ordinary squares and antisquares, this infinite binary word is 3^+-parameterized-square-free. Moreover, every order-preserving square is a parameterized square, and hence the word is 3^+-order-preserving-square-free. Their results were proved using large morphisms found through computer search and `Walnut` [15]. Combining these results gives a complete characterization of the values of ℓ and the alphabet sizes for which infinite ℓ^+-square-free words exist in the parameterized and order-preserving models.

2 Preliminaries

Let Σ be an *alphabet*. We denote the size of Σ by σ. An element of Σ is referred to as a *character*, and an element of Σ^* is referred to as a *word*. We denote a word x of length n as $x = x[0] \cdots x[n-1]$, where $x[i]$ denotes the i-th character of x. For $0 \leq i < j \leq |x|$, we denote by $x[i..j)$ the subword of x consisting of the characters $x[i] \cdots x[j-1]$, where the notation $[i..j)$ follows the half-open interval convention. If a word x does not contain another word w as a subword, we say that x *avoids* w.

Two words $x, y \in \Sigma^n$ are *parameterized equivalent* if there exists a bijective function $f : \Sigma \to \Sigma$ such that $f(x[i]) = y[i]$ for all $0 \leq i < n$. Two words $x, y \in \Sigma^n$ on a totally ordered alphabet Σ are *order-preserving equivalent* if $x[i] \preceq x[j] \Leftrightarrow y[i] \preceq y[j]$ holds for every pair $i, j \in [0..n-1]$. We note that order-preserving equivalence is stronger than parameterized equivalence; in particular, if two words x and y are order-preserving equivalent, then they are also parameterized equivalent.

A word of the form ww is called a (standard) *square*. A word that contains no standard square as a subword is called *square-free*. For an integer ℓ, we call such a square an ℓ^+-*square* if its total length is at least 2ℓ. A word of the form uv, where u and v are *parameterized* (resp. *order-preserving*) *equivalent*, is called a *parameterized* (resp. *order-preserving*) *square*. We use the term ℓ^+-*parameterized* (resp. ℓ^+-*order-preserving*) *square* to refer to such a square when its length is at least 2ℓ. A word is said to be ℓ^+-*square-free*, with respect to an equivalence model (exact, parameterized, order-preserving, etc.), if it contains no square of the corresponding type with length at least 2ℓ.

A *morphism* on two alphabets Σ and Σ' is a function $h : \Sigma^* \to \Sigma'^*$ such that $h(xy) = h(x)h(y)$ for all words $x, y \in \Sigma^*$. Hence, to define a morphism h, it suffices to specify $h(c)$ for each character $c \in \Sigma$. When $\Sigma' = \Sigma$ (i.e., $h : \Sigma^* \to \Sigma^*$), we define $h^k(x)$ for $x \in \Sigma^*$ as the word obtained by applying h to x repeatedly k times. If x is a prefix of $h(x)$, then $h^k(x)$ is a prefix of $h^{k+1}(x)$ for every $k \geq 0$. If the lengths $|h^k(x)|$ are unbounded, these finite words define an infinite word. Let $\Sigma_2 = \{0, 1\}$ and $\Sigma_3 = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$ denote the binary and ternary alphabets, respectively. In this paper, we put special emphasis on the morphism $\phi : \Sigma_3^* \to \Sigma_3^*$ defined by

$$\begin{aligned} \mathrm{a} &\mapsto \mathrm{abc}, \\ \mathrm{b} &\mapsto \mathrm{ac}, \\ \mathrm{c} &\mapsto \mathrm{b}. \end{aligned}$$

It is known that $\phi^k(\mathrm{a})$ is square-free for any $k \geq 0$ [3]. For instance, $\phi(\mathrm{a}) = \mathrm{abc}$, $\phi^2(\mathrm{a}) = \mathrm{abcacb}$, and $\phi^3(\mathrm{a}) = \mathrm{abcacbabcbac}$, none of which contains a square.

3 3^+-Parameterized-Square-Free Ternary Words

In this section, we aim to construct an infinite 3^+-parameterized-square-free ternary word. We define $C = \mathrm{cbbbc}$ and let $\mu : \Sigma_3^* \to \Sigma_3^*$ be the morphism that maps c to C while leaving a and b unchanged. For each $k \geq 0$, we define $x'_k = \phi^k(\mathrm{a})$ and $x_k = \mu(x'_k)$. Recall that x'_k is square-free [3].

We first show that x_k contains no 3^+-parameterized squares except for the standard squares.

Lemma 1. *For any $k \geq 0$, if a 3^+-parameterized square $w = uv$ ($|u| = |v| \geq 3$) exists as a subword of x_k, then $u = v$.*

Proof. Let $w = uv$ be a 3^+-parameterized square subword of x_k of length 2ℓ ($\ell \geq 3$), where u and v are parameterized equivalent under some bijection f.

We first determine the nonempty subwords of x'_k that avoid c. Since every subword of the square-free word x'_k is square-free, such a subword cannot contain any of the binary squares aa, bb, abab, and baba. Indeed, avoiding aa and bb forces a binary word to alternate, and every alternating word of length at least four contains abab or baba. Thus, any nonempty subword of x'_k that avoids c belongs to $\{\mathrm{a}, \mathrm{b}, \mathrm{ab}, \mathrm{ba}, \mathrm{aba}, \mathrm{bab}\}$. We next exclude aba. This is clear for $x'_0 = \mathrm{a}$. For $k \geq 1$, write $x'_k = \phi(z)$ with $z = x'_{k-1}$. In any occurrence of aba, the middle b is preceded and followed by a. Consider an occurrence of b in $\phi(z)$. It is either the middle character of $\phi(\mathrm{a}) = \mathrm{abc}$ or the whole image $\phi(\mathrm{c}) = \mathrm{b}$, since $\phi(\mathrm{b}) = \mathrm{ac}$ contains no b. In the first case, this b is followed by c, and hence it cannot be the middle character of aba. In the second case, if this c is the first character of z, then this b has no preceding character and cannot be the middle character of aba. Otherwise, the character preceding this b is the last character of $\phi(d)$ for the character d preceding this c in z. Since $\phi(\mathrm{a}) = \mathrm{abc}$, $\phi(\mathrm{b}) = \mathrm{ac}$, and $\phi(\mathrm{c}) = \mathrm{b}$ end with c, c, and b, respectively, the character preceding this b is not a. Hence aba does not occur in x'_k. Consequently, every nonempty subword of x'_k that avoids c belongs to $S = \{\mathrm{a}, \mathrm{b}, \mathrm{ab}, \mathrm{ba}, \mathrm{bab}\}$.

We next show that bb occurs in w. Since x_k is obtained from x'_k by replacing each occurrence of c with C, any subword of x_k that avoids bb cannot contain an occurrence of C, because $C = \mathrm{cbbbc}$ contains bb. Moreover, no suffix or prefix of C of length at least 3 avoids bb. Suppose, for the sake of contradiction, that w avoids bb. Then, it must be of the form $w = \alpha s \beta$, where α (resp. β) is a suffix (resp. prefix) of C of length at most 2 and $s \in S$. This gives $|s| = 2\ell - |\alpha| - |\beta| \geq 2$, so the only possibilities for s are ab, ba, and bab. Consequently, the only possible candidates for w, whose length is even and at least six, are bcbabc, cbabcb, bcabcb, and bcbacb. None of them is a parameterized square. Hence, bb occurs in w.

Since $|w| \geq 6$ and every occurrence of bb in x_k must be a part of an image of c (i.e., $C = \mathrm{cbbbc}$), either u or v must contain the subword cbb or bbc. Since x'_k is

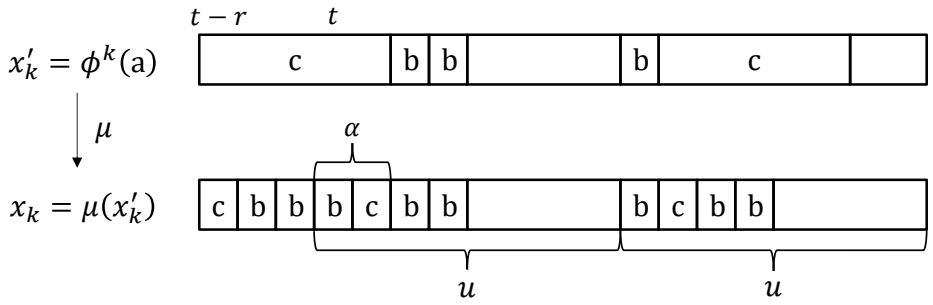

Figure 1. Illustration for Lemma 2. In this figure, the two occurrences of u in $w = u^2$ have different offsets. The left one has offset $r = 3$, and the right one starts at a boundary, that is, with offset 0. In this case, u starts with αbb = bcbb, where α = bc. The factorization starting at position $t - r$ is C|b|b. This implies that x'_k contains the square bb.

square-free, it contains no two equal consecutive characters. The morphism μ only replaces c with C = cbbbc. Thus, every occurrence of two equal consecutive characters in x_k is contained in the block bbb of an occurrence of C. Hence, if a length-three subword of x_k has its last two (resp. first two) characters equal and the remaining character different, then it must be cbb (resp. bbc). Thus, the only subword of x_k parameterized equivalent to cbb (resp. bbc) is cbb itself (resp. bbc itself). Therefore, the corresponding subword in the other half of w is identical to cbb or bbc, and hence f(b) = b and f(c) = c. Consequently, f must be the identity function, and w is a standard square $w = uu$. □

We also show the 3^+-square-freeness of x_k.

Lemma 2. *For any $k \geq 0$, the word x_k is 3^+-square-free.*

Proof. Let $\mathcal{S} = \{\mathrm{a}, \mathrm{b}, C\}$. In this proof, we define a *factorization* of a word z as a decomposition $z = f_1 \cdots f_m$ with $f_i \in \mathcal{S}$ for every i. The words f_i are called *factors*. Since $\mathcal{S}$ is prefix-free, such a factorization is unique whenever it exists. Moreover, since μ maps c to C while leaving a and b unchanged, the word $x_k = \mu(x'_k)$ has a factorization, and hence it has a unique factorization. We call a position a *boundary* if a factor starts there. If a position is not a boundary, then it lies inside an occurrence of C. In this case, we denote its offset from the beginning of that occurrence to be an element of $\{1, 2, 3, 4\}$. Boundary positions are said to have offset 0. If both endpoints of a subword z of x_k are boundaries, then its factorization determines a unique word v such that $\mu(v) = z$.

For the sake of contradiction, assume that there exists a square subword $w = uu = x_k[p - \ell..p + \ell)$ of x_k with $\ell \geq 3$ centered at text position p. Let $i = p - \ell$ be the starting position of the square.

We first show that the two occurrences $x_k[i..p)$ and $x_k[p..p + \ell)$ of u start at the same offset. Assume that one occurrence starts at offset $r \in \{1, 2\}$. Then it begins with bbb if $r = 1$ and with bbc if $r = 2$. These prefixes occur only inside an occurrence of C, and the offset is uniquely determined. Hence the other occurrence also starts at the same offset r.

Next assume that one occurrence starts at offset $r \in \{3, 4\}$. Let $\alpha = C[r..5)$, so α = bc for $r = 3$ and α = c for $r = 4$. If an occurrence starts away from a boundary, then the prefix α determines its offset uniquely. Suppose, for the sake of contradiction, that the other occurrence starts at a different offset. Since the prefix α determines every non-boundary offset, the other occurrence must start at a boundary. In this

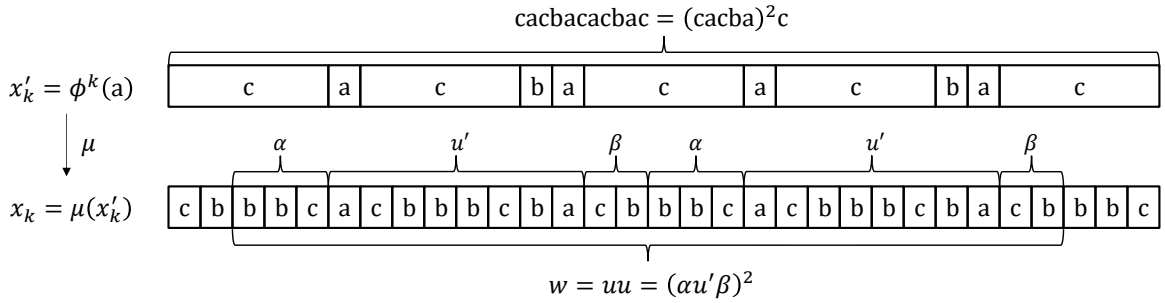

Figure 2. Illustration for Lemma 2. In the figure, x_k has a square $w = uu = (\mathrm{bbcacbbbcbacb})^2$ and u is written as $u = \alpha z \beta$ where $\alpha = \mathrm{bbc}, \beta = \mathrm{cb}$, and $z = \mathrm{acbbbcba}$. In this case, $w = uu$ is generated from $(\mathrm{cacba})^2 c$ and thus x'_k has a square $(\mathrm{cacba})^2$.

case, the other occurrence of u starts at offset 0 with prefix α. If $r = 4$ ($\alpha = \mathrm{c}$), then the first factor in the factorization beginning at this occurrence must be C. If $r = 3$ ($\alpha = \mathrm{bc}$), then the first and second factors in this factorization must be b and C, respectively. In both cases, u starts with $\alpha\mathrm{bb}$. Consider the occurrence of u with offset r, and let its starting position be t. Since α appears as a suffix of the occurrence of C starting at position $t - r$, the factorization of x_k at position $t - r$ begins with $C|\mathrm{b}|\mathrm{b}$. This factorization implies that x'_k contains the square bb, contradicting the square-freeness of x'_k (see Figure 1). Thus, whenever one occurrence of u starts away from a boundary, the other occurrence starts at the same offset in $\{1, 2, 3, 4\}$. Equivalently, if one occurrence starts at a boundary, then the other occurrence also starts at a boundary.

If both occurrences of u start at boundaries, then both endpoints of the first occurrence $x_k[i..p)$ are boundaries. The equality of the two occurrences and the uniqueness of the factorization imply that $x_k[p..p + \ell)$ has the same factorization. Thus, both endpoints of $x_k[p..p + \ell)$ are also boundaries. Let v be the unique word such that $\mu(v) = x_k[i..p)$. Then both endpoints of w are boundaries, and $w = \mu(v^2)$. Hence v^2 occurs in x'_k. This contradicts the square-freeness of x'_k. Otherwise, let $r \in \{1, 2, 3, 4\}$ be the common offset of the two occurrences of u. Let α be the suffix of C of length $5 - r$ and β be the prefix of C of length r. Since $\beta\alpha = C$ and the center position p of the square $w = uu$ lies on an occurrence of C with offset $r = |\beta|$, u can be written as $u = \alpha z \beta$, where $z = x_k[i + |\alpha|..p - |\beta|)$ (see Figure 2). Since the positions $i + |\alpha|$, $p - |\beta|$, $p + |\alpha|$, and $p + \ell - |\beta|$ are boundaries, the two subwords $x_k[i + |\alpha|..p - |\beta|)$ and $x_k[p + |\alpha|..p + \ell - |\beta|)$ have exactly the same factorization. Let v be the unique word such that $\mu(v) = z$. Hence, $w = \alpha\mu(v)\beta\alpha\mu(v)\beta$. Since both the starting and ending position of w lie inside occurrences of C, by extending this subword to the left by r positions and to the right by $5 - r$ positions, we obtain $C\mu(v)C\mu(v)C$ as a subword of x_k. This subword is equal to $\mu(\mathrm{cvcvc})$. Thus, x'_k contains the square $(\mathrm{cv})^2$. This contradicts the square-freeness of x'_k. □

By combining the above results, we can obtain the main result.

Theorem 3. *For any $k \geq 0$, the ternary word $x_k = \mu(\phi^k(\mathrm{a}))$ is 3^+-parameterized-square-free.*

Proof. By Lemma 1, if a subword w is a 3^+-parameterized square, then w must be a standard square. However, by Lemma 2, such square does not exist. Therefore x_k does not contain any 3^+-parameterized-square. □

4 3^+-Order-Preserving-Square-Free Binary Words

In this section, we show that there exists an infinite 3^+-order-preserving-square-free binary word. The proof is similar to [19], where it is observed that applying the morphism $\psi : \Sigma_3^* \to \Sigma_2^*$ with $\psi(\mathrm{a}) = 0000, \psi(\mathrm{b}) = 0101$, and $\psi(\mathrm{c}) = 1111$ to a square-free ternary word x yields a 3^+-square-free binary word (in the strict sense). We show that the same property holds when we slightly modify the morphism ψ to another morphism $\tau : \Sigma_3^* \to \Sigma_2^*$ (Lemma 4). Then, we construct an infinite binary word by combining the two morphisms τ and ϕ (Theorem 5).

Let $\tau : \Sigma_3^* \to \Sigma_2^*$ be the morphism defined as

$$\begin{aligned} \mathrm{a} &\mapsto 00, \\ \mathrm{b} &\mapsto 0101, \\ \mathrm{c} &\mapsto 11. \end{aligned}$$

The following lemma will be proved in Section 4.1.

Lemma 4. *For any square-free ternary word x, the word $\tau(x)$ is 3^+-square-free.*

Let $x'_k = \phi^k(\mathrm{a})$ and $y_k = \tau(x'_k)$ for every $k \geq 0$. Recall that each x'_k is square-free, and thus, each y_k is 3^+-square-free by Lemma 4. We prove the main theorem of this section.

Theorem 5. *For any $k \geq 0$, the binary word $y_k = \tau(\phi^k(\mathrm{a}))$ is 3^+-order-preserving-square-free.*

Proof. Assume, for the sake of contradiction, that there exists a subword $w = uv$ ($|u| = |v|$) of y_k that is a 3^+-order-preserving square of length 2ℓ ($\ell \geq 3$). Since y_k is 3^+-square-free and $\ell \geq 3$, we have $u \neq v$. If either u or v contains both 0 and 1, then the only order-preserving bijection between their alphabets is the identity. Therefore, since $u \neq v$, the only possible case is when both u and v consist of a single repeated character but differ from each other (i.e., $\{u, v\} = \{0^\ell, 1^\ell\}$). To prove that such a square cannot occur in y_k, it suffices to show that y_k contains neither $0^3 1^3$ nor $1^3 0^3$ as a subword.

Since $\tau(\mathrm{a}) = 00$, $\tau(\mathrm{b}) = 0101$, and $\tau(\mathrm{c}) = 11$, any occurrence of 0^3 (resp. 1^3) in $y_k = \tau(x'_k)$ is contained in $\tau(\mathrm{aa})$ or $\tau(\mathrm{ab})$ (resp. $\tau(\mathrm{cc})$ or $\tau(\mathrm{bc})$), if it exists. Since x'_k is square-free, it contains neither aa nor cc. Thus, 0^3 can appear only as the prefix of $\tau(\mathrm{ab}) = 000101$, and 1^3 can appear only as the suffix of $\tau(\mathrm{bc}) = 010111$. Therefore, a subword of the form $0^3 1^3$ or $1^3 0^3$ can occur only inside $\tau(\mathrm{bcab})$, so that

$$\tau(\mathrm{bcab}) = \tau(\mathrm{b})\tau(\mathrm{c})\tau(\mathrm{a})\tau(\mathrm{b}) = 010\,1^3 0^3\,101.$$

However, bcab cannot occur in x'_k. If $k = 0$, then this is immediate from $x'_0 = \mathrm{a}$. If $k \geq 1$, then both bc and ab can only be generated by applying ϕ to the character a. Thus, an occurrence of bcab in $x'_k = \phi(x'_{k-1})$ would imply an occurrence of aa in x'_{k-1}, contradicting the square-freeness of x'_{k-1}. Therefore, y_k contains no 3^+-order-preserving square. □

In the rest of this section, we give a proof of Lemma 4.

4.1 Proof of Lemma 4

Before starting the proof, we introduce some notation. Let x be a square-free ternary word. We denote $w = \tau(x)$. A subword $w[i..j)$ of w is called a *block* if it is exactly generated from the subword $x[s..e)$, i.e., there exists a pair of integers s and e such that $\tau(x[0..s)) = w[0..i)$ and $\tau(x[s..e)) = w[i..j)$ hold. We denote $\mathcal{F} = \{\tau(\mathrm{a}), \tau(\mathrm{b}), \tau(\mathrm{c})\} = \{00, 0101, 11\}$. In this subsection, we define a *factorization* of a block z as a decomposition $z = f_1 \cdots f_m$ with $f_i \in \mathcal{F}$ for every i. The words f_i are called *factors*. Note that this convention differs from the factorization used in Lemma 2. Since $\mathcal{F}$ is prefix-free, such a factorization is unique whenever it exists. Every block $w[i..j)$ has such a factorization by definition, and therefore its factorization is uniquely defined. Moreover, the factorization of a block z determines a unique word v such that $\tau(v) = z$. A position i with $0 \le i \le |w|$ is called a *boundary* if $i = |w|$ or a factor starts at position i in the factorization of w. For any position i, let o_i and o'_i be the smallest non-negative integers such that $i + o_i$ and $i - o'_i$ are boundaries, respectively, where $0 \le o_i, o'_i \le 3$. In particular, $o_i = 0$ if and only if $o'_i = 0$. Since the length of a factor is either two (for 00, 11) or four (for 0101), $o_i \ge 2$ implies that the factor ending at position $i + o_i$ is 0101. Similarly, $o'_i \ge 2$ implies that the factor starting at position $i - o'_i$ is 0101.

Next, we introduce Fact 4.1, which states that w contains no short squares of lengths between 6 and 12. Let $w[i..j)$ be a subword of w of length 2ℓ. By construction of w, the subword $w[i - o'_i..j + o_j)$ is a block generated from a square-free subword of x of length at most $\ell + 1$. Indeed, each factor has length at least two, so a subword of length 2ℓ can intersect at most $\ell + 1$ factors. Thus, we can verify Fact 4.1 by enumerating all words in $\{\tau(v) \mid v \in \Sigma_3^*, 1 \le |v| \le 7, v \text{ is square-free}\}$ and checking 3^+-square-freeness[1].

Fact: For every $3 \le \ell \le 6$, the word w contains no square of length 2ℓ.

We next establish two lemmas, which rule out long equal subwords with different boundary offsets.

Lemma 6. *There is no pair of subwords $w[i..i + \ell)$ and $w[j..j + \ell)$ such that $\ell \ge 7$, $w[i..i + \ell) = w[j..j + \ell)$, and $o_i \not\equiv o_j \pmod 2$.*

Proof. Assume on the contrary that there are two subwords $w[i..i + \ell) = w[j..j + \ell)$ of length $\ell \ge 7$ with $o_i \not\equiv o_j \pmod 2$. Let $u = w[i..i + \ell) = w[j..j + \ell)$. Assume $o_i < o_j$ without loss of generality. We first consider the case $o_i = 0, o_j = 3$. In this case, $w[j - 1..j + 3)$ must be a factor, specifically $w[j - 1..j + 3) = 0101$. Then, $w[j..j + 3) = w[i..i + 3) = 101$, which contradicts $o_i = 0$ since no factor starts with 101.

The remaining case is $o_j = o_i + 1$. Let f_i and f_j be the factors starting at positions $i + o_i$ and $j + o_j$, respectively. Since $o_j = o_i + 1$, we have $0 \le o_i \le 2$. Together with $\ell \ge 7$ and the fact that every factor has length at most four, this ensures that the factor immediately following f_i exists. Let g_i be this factor. Since $\ell \ge 7$, both f_i and f_j are contained in $u = w[i..i + \ell) = w[j..j + \ell)$. Further, since $o_j = o_i + 1$, the starting positions of the occurrences of f_i and f_j inside u are different by one position. These facts imply that $(f_i g_i)[1..|f_j| + 1) = f_j$ holds.

We consider the following three cases according to the value of f_i.

Case 1: $f_i = 0101$: Then f_j must start with 10 because $f_j[0..2) = u[o_j..o_j + 2) = u[o_i + 1..o_i + 3) = f_i[1..3) = 10$. However, no factor starts with 10, a contradiction.

[1] The source code is available at `https://github.com/koeppl/squarechecker/check_short_string_3plus_square_freeness.py`

Case 2: $f_i = 11$: Then f_j must start with 1, hence $f_j = 11 \in \mathcal{F}$. From $(f_i g_i)[1..|f_j|+1) = f_j = 11$, g_i also starts with 1, so $g_i = 11$. Therefore, $f_i g_i = 1111$, which is the image of cc under τ. This implies that x contains the square cc, contradicting the square-freeness of x.
Case 3: $f_i = 00$: Then f_j must start with 0, so f_j is either 00 or 0101. If $f_j = 00$, as in the previous case, we obtain $g_i = 00$ from $(f_i g_i)[1..|f_j|+1) = f_j = 00$. Then, $f_i g_i = 0000 = \tau(\mathrm{aa})$, contradicting the square-freeness of x. Otherwise, if $f_j = 0101$, then g_i must start with 10, which contradicts that $g_i \in \mathcal{F}$. □

Lemma 7. *There is no pair of subwords $w[i..i+\ell)$ and $w[j..j+\ell)$ such that $\ell \geq 7$, $w[i..i+\ell) = w[j..j+\ell)$, and $o_i - o_j \equiv 2 \pmod 4$.*

Proof. Assume on the contrary that there are two subwords $w[i..i+\ell) = w[j..j+\ell)$ of length $\ell \geq 7$ with $o_i - o_j \equiv 2 \pmod 4$. Let $u = w[i..i+\ell) = w[j..j+\ell)$. Assume $o_i < o_j$ without loss of generality. Then, the only possible pairs for (o_i, o_j) are $(0,2)$ and $(1,3)$, so $o_j = o_i + 2$. Let f_i and f_j be the factors starting at positions $i + o_i$ and $j + o_j$, respectively. Also, let e_j be the factor immediately preceding f_j. Since $o_j \geq 2$, we have $e_j = 0101$.

From $u = w[i..i+\ell) = w[j..j+\ell)$ and $o_j = o_i + 2$, it follows that $u[o_i..o_i+2) = u[o_j - 2..o_j) = e_j[|e_j| - 2..|e_j|) = 01$. Hence $f_i = 0101$, because it is the only factor that starts with 01. Also, $u[o_j..o_j+2) = u[o_i+2..o_i+4) = f_i[2..4) = 01$, so $f_j = 0101$. Therefore, the two consecutive factors around position $j + o_j$ satisfy $e_j = f_j = 0101$. Consequently, $e_j f_j = 01010101 = \tau(\mathrm{bb})$, which implies that x contains the square bb, contradicting the square-freeness of x. □

Now, we are ready to prove Lemma 4.

Proof (of Lemma 4). For the sake of contradiction, assume w contains a square $w[i..k)$ of length $2\ell \geq 6$. Let $j = i + \ell$ where $\ell = (k-i)/2$, so $w[i..k) = uu$ with $u = w[i..j) = w[j..k)$. By Fact 4.1, w has no square with $3 \leq \ell \leq 6$, thus $\ell \geq 7$. Let $d = (o_i - o_j) \bmod 4$. We classify squares in w by the values of o_i, o'_j, o_j, o'_k and d.
Case 1: $d = 2$: This case contradicts Lemma 7.
Case 2: $d \in \{1, 3\}$: This case is equivalent to $o_i \not\equiv o_j \pmod 2$. Thus, the existence of square $w[i..k) = uu$ contradicts Lemma 6.
Case 3: $d = 0$: In this case, $o_i = o_j$ holds. Assume $w[i..j)$ is factorized as $w[i..j) = \alpha f_1 \cdots f_t \beta$, where $\alpha = w[i..i+o_i)$ and $\beta = w[j-o'_j..j)$ may be empty, and each f_p is a factor. Then, $w[j..k)$ is also factorized as $w[j..k) = \alpha f_1 \cdots f_t \beta$, because $w[i+o_i..j) = w[j+o_j..k)$, both $i+o_i$ and $j+o_j$ are boundaries, and the factorization of a block is unique. More precisely, $o'_j = o'_k$ and $w[i..k) = \alpha f_1 \cdots f_t \beta \alpha f_1 \cdots f_t \beta$ hold. We denote $f = f_1 \cdots f_t$, and let g be the word such that $\tau(g) = f$. Note that such g is uniquely defined.

Case 3-(a): $o_i = o_j = 0$: In this case, both i and j are boundaries. Hence, both α and β are the empty word, and $w[i..j)$ is fully factorized as $w[i..j) = f_1 \cdots f_t = f$. Likewise, $w[j..k) = f$, and thus $w[i..k) = w[i..j)w[j..k) = f^2$. Since $\tau(g^2) = f^2$, the block $w[i..k)$ is generated from the word g^2. Hence g^2 occurs in x, a contradiction.

Case 3-(b): $o_i = o_j \in \{2, 3\}$: In this case, the factors crossing positions i and j must be 0101. In particular, $w[i-o'_i..i+o_i) = w[j-o'_j..j+o_j) = 0101$. Thus, the block $w[i-o'_i..k-o'_k)$ is factorized as $|0101|f|0101|f|$. Therefore, this block is generated from the word $(\mathrm{b}g)^2$, implying that x contains a square (see the top of Figure 3), a contradiction.

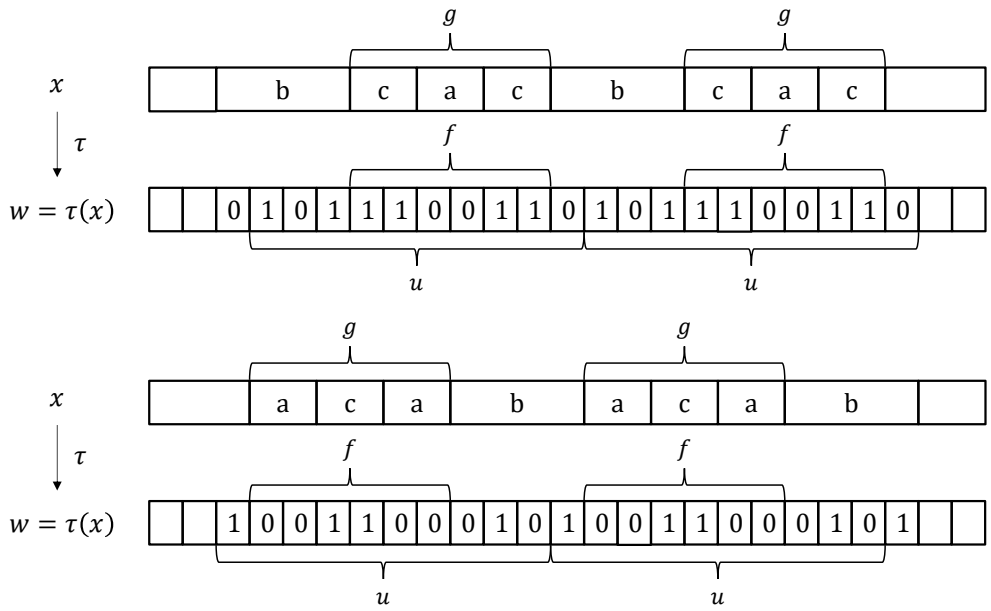

Figure 3. Illustrations of examples for Case 3-(b) (top) and Case 3-(c)-(ii) (bottom) of Lemma 4. In the first figure, the word $w = \tau(x)$ contains a square $uu = (1011100110)^2$ and x contains a square $(\mathrm{b}g)^2 = (\mathrm{bcac})^2$. In the second figure, the word $w = \tau(x)$ contains a square $uu = (1001100010)^2$ and x contains a square $(g\mathrm{b})^2 = (\mathrm{acab})^2$.

Case 3-(c)-(i): $o_i = o_j = 1$ and $o'_j = o'_k = 1$: In this case, the factor crossing position j has length two, i.e., $w[j-o'_j..j+o_j)$ is either 00 or 11. Let c be the character such that $w[j - o'_j..j + o_j) = cc$. Since $w[j - o'_j]$ and $w[j]$ correspond to the last and first characters of u, respectively, we have $\alpha = \beta = c$. Therefore, $w[i..k) = uu$ is of the form $cfccfc$. If $c = 0$, the factor crossing position i must be 00, because it is the only factor that ends with 0. Then, the block $w[i - o'_i..k - o'_k)$ is factorized as $|00|f|00|f|$. Therefore, this block is generated from the word $(\mathrm{a}g)^2$, implying that x contains a square (see the top of Figure 4), a contradiction. If $c = 1$, the factor crossing position k must be 11, because it is the only factor that starts with 1. Then, the block $w[i+o_i..k+o_k)$ is factorized as $|f|11|f|11|$. Therefore, this block is generated from the word $(g\mathrm{c})^2$, implying that x contains a square (see the bottom of Figure 4), a contradiction.

Case 3-(c)-(ii): $o_i = o_j = 1$ and $o'_j = o'_k \neq 1$: In this case, $o'_j = o'_k = 3$. This is because $o_j = 1$, so j is not a boundary and the factor crossing position j has length $o_j + o'_j = 1 + o'_j$. Every factor has length either 2 or 4. The proof of this case is analogous to that of Case 3-(b). In this case, the factors crossing positions j and k are 0101, so the block $w[i + o_i..k + o_k)$ is factorized as $|f|0101|f|0101|$. Therefore, this block is generated from the word $(g\mathrm{b})^2$, implying that x contains a square (see the bottom of Figure 3), a contradiction.

The preceding cases exhaust all possible scenarios for a square uu occurring in $w = \tau(x)$. Since each of these cases leads to a contradiction with the square-freeness of x, we conclude that $w = \tau(x)$ is 3^+-square-free. □

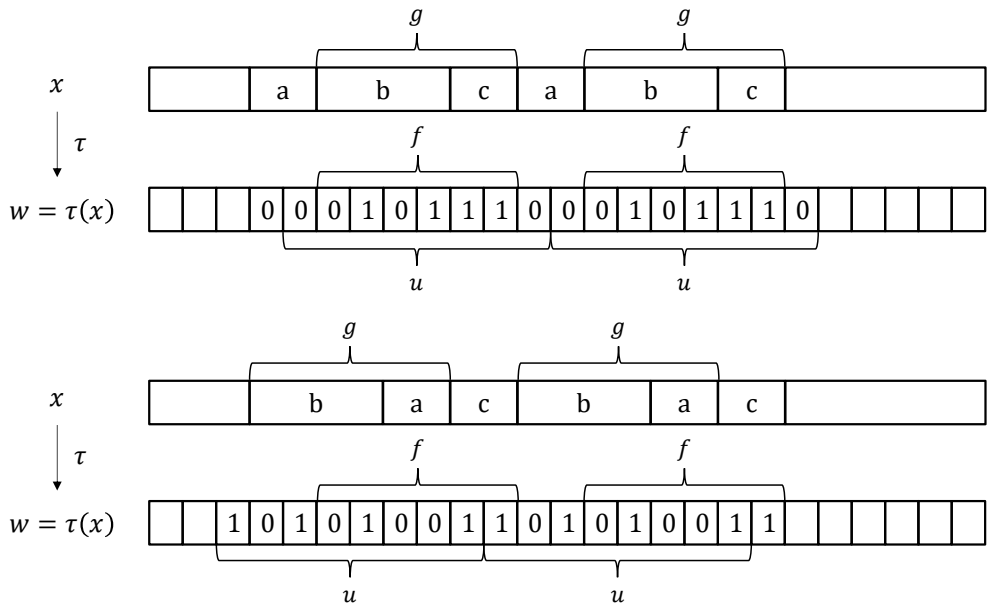

Figure 4. Illustrations of examples for Case 3-(c)-(i) of Lemma 4. In the first figure, the word $w = \tau(x)$ contains a square $uu = (00101110)^2$ and x contains a square $(\mathrm{a}g)^2 = (\mathrm{abc})^2$. In the second figure, the word $w = \tau(x)$ contains a square $uu = (10101001)^2$ and x contains a square $(g\mathrm{c})^2 = (\mathrm{bac})^2$.

5 Conclusions

In this paper, we studied ℓ^+-square-free words under several equivalence relations. We gave combinatorial proofs of the existence of an infinite 3^+-parameterized-square-free ternary word and an infinite 3^+-order-preserving-square-free binary word. Both infinite words were constructed using small morphisms.

Through computational experiments, we also determined the maximum lengths in the finite cases reported in Table 1. Table 2 gives an explicit longest word for each such case. These values and words were obtained with software tools[2]. We list alphabet sizes only up to $\sigma = 5$. For the finite entries in the tables, our experiments also gave the same values when $\sigma \geq 6$.

As a direction for future work, it would be interesting to study repetition thresholds [7,1] under these equivalence relations. A repetition threshold is the smallest real number r such that some infinite word avoids every factor of exponent greater than r. This can be viewed as another direction in generalizing square-freeness: while ℓ^+-square-freeness relaxes classical square-freeness by considering only long squares, repetition thresholds relax it by bounding the exponent of repetitions. In the strict-equality setting, repetition thresholds have been studied extensively, including questions about the maximum exponent and the number of occurrences of factors attaining it. Whether analogous results hold under other equivalence relations is an interesting question.

Acknowledgement. We thank Pascal Ochem for pointing out relevant prior work. This work was supported by JSPS KAKENHI Grant Numbers JP24K20734 (TM),

[2] `https://github.com/koeppl/squarechecker`

Table 2. Examples of longest ℓ^+-square-free words under various equivalence relations and alphabet sizes. Each finite entry gives one longest word. An entry marked ∞ indicates the existence of an infinite ℓ^+-square-free word. A waved line indicates a conjectured result based on experimental results. A dash ($-$) indicates that infiniteness follows from another case (e.g., a smaller ℓ or σ).

Strict Equality

$\sigma\backslash\ell$	1	2	3	4
2	aba	abaabbaaabbbaabbab	∞	$-$
3	∞	$-$	$-$	$-$
4	$-$	$-$	$-$	$-$
5	$-$	$-$	$-$	$-$

Parameterized Equivalence

$\sigma\backslash\ell$	1	2	3	4
2	a	aaabaaa	∞	$-$
3	a	aabaaabcc	$-$	$-$
4	a	aabaaabcc	$-$	$-$
5	a	aabaaabcc	$-$	$-$

Order-Preserving Equivalence

$\sigma\backslash\ell$	1	2	3	4
2	a	aaabaaa	∞	$-$
3	a	∞	$-$	$-$
4	a	$-$	$-$	$-$
5	a	$-$	$-$	$-$

Cartesian-Tree Equivalence

$\sigma\backslash\ell$	1	2	3	4
2	a	baaba	bbaaabbabaabbbababaaabbababaa	$\approx\infty$
3	a	aacbaccba	$\approx\infty$	$-$
4	a	aacbaccba	$-$	$-$
5	a	aacbaccba	$-$	$-$

JP25K21150 (DK), JP23K24808, JP23K18466 (SI), and JST BOOST, Japan Grant Number JPMJBS2406 (HS).

References

1. G. Badkobeh, M. Crochemore, and M. Rao: *Finite repetition threshold for large alphabets.* RAIRO Theor. Informatics Appl., 48(4) 2014, pp. 419–430.
2. B. S. Baker: *Parameterized pattern matching: Algorithms and applications.* J. Comput. Syst. Sci., 52(1) 1996, pp. 28–42.
3. R. Cori and M. R. Formisano: *Partially abelian squarefree words.* RAIRO Theor. Informatics Appl., 24 1990, pp. 509–520.
4. M. Crochemore and W. Rytter: *Squares, cubes, and time-space efficient string searching.* Algorithmica, 13(5) 1995, pp. 405–425.
5. J. D. Currie and J. T. Johnson: *Characterization of the lengths of binary circular words containing no squares other than 00, 11, and 0101.* Theor. Comput. Sci., 850 2021, pp. 30–39.
6. J. D. Currie, F. Manea, D. Nowotka, and K. Reshadi: *Unary patterns under permutations.* Theor. Comput. Sci., 743 2018, pp. 72–82.
7. F. Dejean: *Sur un théorème de thue.* J. Comb. Theory A, 13(1) 1972, pp. 90–99.
8. R. C. Entringer, D. E. Jackson, and J. A. Schatz: *On nonrepetitive sequences.* J. Comb. Theory A, 16(2) 1974, pp. 159–164.
9. A. S. Fraenkel and J. Simpson: *How many squares can a string contain?* J. Comb. Theory, Ser. A, 82(1) 1998, pp. 112–120.
10. R. Hamai, K. Taketsugu, Y. Nakashima, S. Inenaga, and H. Bannai: *On the number of non-equivalent parameterized squares in a string*, in String Processing and Information Retrieval - 31st International Symposium, SPIRE 2024, Puerto Vallarta, Mexico, September 23-25, 2024, Proceedings, Z. Lipták, E. S. de Moura, K. Figueroa, and R. Baeza-Yates, eds., vol. 14899 of Lecture Notes in Computer Science, Springer, 2024, pp. 174–183.
11. V. Keränen: *Abelian squares are avoidable on 4 letters*, in Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria, July 13-17, 1992, Proceedings, vol. 623 of Lecture Notes in Computer Science, Springer, 1992, pp. 41–52.
12. J. Kim, P. Eades, R. Fleischer, S. Hong, C. S. Iliopoulos, K. Park, S. J. Puglisi, and T. Tokuyama: *Order-preserving matching.* Theor. Comput. Sci., 525 2014, pp. 68–79.
13. T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen: *Maximum number of distinct and nonequivalent nonstandard squares in a word.* Theor. Comput. Sci., 648 2016, pp. 84–95.
14. F. Manea, M. Müller, and D. Nowotka: *Cubic patterns with permutations.* J. Comput. Syst. Sci., 81(7) 2015, pp. 1298–1310.
15. H. Mousavi: *Automatic theorem proving in walnut.* CoRR, abs/1603.06017 2016.
16. T. Ng, P. Ochem, N. Rampersad, and J. O. Shallit: *New results on pseudosquare avoidance*, in Combinatorics on Words - 12th International Conference, WORDS 2019, Loughborough, UK, September 9-13, 2019, Proceedings, R. Mercas and D. Reidenbach, eds., vol. 11682 of Lecture Notes in Computer Science, Springer, 2019, pp. 264–274.
17. S. G. Park, A. Amir, G. M. Landau, and K. Park: *Cartesian tree matching and indexing*, in Proc. CPM, vol. 128 of LIPIcs, 2019, pp. 16:1–16:14.
18. H. Prodinger: *Non-repetitive sequences and Gray code.* Discret. Math., 43(1) 1983, pp. 113–116.
19. H. Prodinger and F. J. Urbanek: *Infinite 0-1-sequences without long adjacent identical blocks.* Discret. Math., 28(3) 1979, pp. 277–289.
20. N. Rampersad, J. O. Shallit, and M. Wang: *Avoiding large squares in infinite binary words.* Theor. Comput. Sci., 339(1) 2005, pp. 19–34.
21. A. Thue: *Über unendliche Zeichenreihen.* Norske Vid. Selsk. Skr. Mat. Nat. Kl., 7 1906, pp. 1–22, Reprinted in *Selected mathematical papers of Axel Thue*, T. Nagell (ed.), Universitetsforlaget, Oslo, 1977, pp. 139–158.

A Lower-Bound Cascade for Lossless Pruning in Levenshtein-Based Contour Matching

Stanisław Piotr Skulimowski

Lublin University of Technology
Lublin
Nadbystrzycka 38D
Poland
s.skulimowski@pollub.pl

Abstract. Contour-based reassembly of fragmented two-dimensional objects requires a large number of Levenshtein comparisons between directional chain-code strings. This paper proposes a lossless pre-filtering strategy based on four lower bounds: string length, symbol-frequency histogram, signed directional turn histogram, and g-gram hit count. Candidates are rejected only when a bound proves that their distance must exceed the threshold; otherwise, exact verification is preserved. Experiments covered seeded synthetic strings, exhaustive whole-space validations, and real Afrosiyab relief contour data. No false negatives were observed in any tested configuration. In representative generated tests, survival decreased to 0.160%, reducing exact verification by 99.840% and yielding a 4.05× speedup. Real-data experiments confirmed strong parameter-dependent pruning under a redundancy-free unordered window-pair evaluation, with histogram filtering dominating equal-window settings.

Keywords: Directional contour strings, Chain-code matching, Approximate string matching, Levenshtein distance, Lower-bound filtering, Lossless pre-filtering, Candidate pruning, Fragment reassembly

1 Introduction

Contour-based reassembly encodes fragment boundaries as symbolic sequences, such as Freeman or extended slope chaincodes, and compares them by approximate string matching. Levenshtein distance models sampling and erosion errors, but standard dynamic programming costs $\Theta(mn)$ for strings of lengths m and n. Large candidate sets therefore motivate filtering [6].

Heuristic filters may be efficient [13], but reconstruction requires recall preservation: false positives increase verification effort, whereas false negatives may remove valid adjacencies [18]. Safe rejection therefore requires a lower bound on the exact distance. Existing counting, partitioning, and g-gram filters may lose selectivity at permissive thresholds [23], and ordinary symbol filters ignore directional geometry [24].

This paper evaluates a candidate-wise cascade of length, symbol-histogram, signed-turn-histogram, and g-gram bounds under static and percentage-based window policies [25]. It combines general string properties with geometry-aware transition profiles and accelerates exhaustive verification over a prescribed candidate set. It is not claimed to outperform every whole-text matcher.

The study investigates three hypotheses:

– **H1 – Losslessness**: The proposed pre-filtering strategy preserves all valid matches obtained by exhaustive Levenshtein verification, and no candidate with distance at most τ is rejected before exact verification.

Stanislaw Piotr Skulimowski: A Lower-Bound Cascade for Lossless Pruning in Levenshtein-Based Contour Matching, pp. 71–84.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

– **H2 – Reduction of exact verification**: The proposed strategy reduces the number of exact Levenshtein computations required to recover the same final match set, as measured by the survival rate p and the candidate reduction rate r.
– **H3 – Parameter-dependent selectivity**: The selectivity and runtime benefit of the filters depend on comparison-window length, edit-distance threshold, alphabet size, retained signature length, gram parameter, dataset type, and active bound set.

The remainder of the paper is organised as follows. The second section describes the context and motivation for pre-filtering research, including the author's apictorial method for arranging elements of flat, decomposed objects, as well as examples of other filters used in text comparison. The third section provides a formal description of the problem and the theoretical assumptions behind the pre-filter operation. The fourth section describes the test environment, dataset, configurations, and metrics used to evaluate the correctness and effectiveness of the filters. The fifth section describes the obtained results and summarises the most important analytical findings. In the sixth section, the author relates the analytical results to the research hypotheses, and in the seventh section, the author summarises the research and presents possible directions for future work.

2 Background and Related Work

The study extends the Skul-Mont apictorial method [19] in which:
– contours of near planar, apictorial objects are encoded as chaincodes using a windrose,
– candidate subcontours (g-grams) are matched according to their Levenshtein distance,
– accepted matches (below the Levenshtein threshold) form a common curve, used for contours' matching through the symmetric difference.

Figure 1 outlines the process.

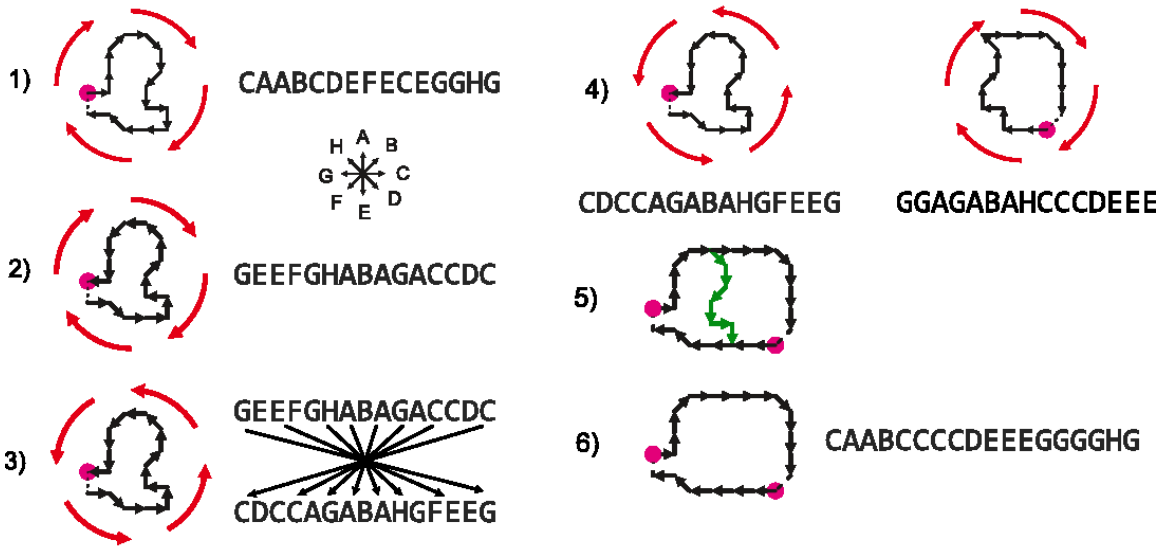

Figure 1. Contour comparison: directional encoding (1), reverse-opposite normalisation of one chaincode (2–3), local g-gram comparison (4), distance-based candidate selection (5), and fragment placement (6).

Exhaustively verifying all windows using the Levenshtein distance and storing all potential connections in the graph is expensive in terms of time and memory. Cost reduction can be achieved in several ways, among others, by reducing the search space before exact verification [4]. Lossless filtering often relies on g-grams, i.e., contiguous substrings of length g. The g-gram lemma provides a necessary lower bound on shared substrings for matching strings [7]. Moreover, g-gram distances are computable in linear time [22], and bag distance compares symbol multiplicities [3]. Ed-Join combines related bounds [23]. Partitioning and automated search schemes extend lossless filtering to larger error budgets [17].

The task is a restricted semi-local comparison: all-semi-local methods compare a complete string against all substrings and compute related prefix–suffix scores [21], whereas this study tests only policy-admissible windows. After the normalisation described in Section 3, direct Shift–Or, BNDM-derived, Myers-style, or semi-global search remains possible.

FASTA and BLAST represent related but algorithmically distinct strategies. FASTA identifies promising regions from short tuple matches and their arrangement along diagonals, whereas BLAST initiates local alignment extension from selected word hits [16,1,2]. Unless their seed condition is necessary whenever $\mathrm{Lev}(X, z) \leq \tau$, they trade sensitivity for speed, as do modern seed systems [20]. They are heuristic candidate generators, whereas this paper studies deterministic rejection over a fixed candidate set.

3 Theoretical Foundations

3.1 Problem Formulation

Let the even directional alphabet[1] be

$$\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{D-1}\}, \tag{1}$$

where $D = |\Sigma|$, and let

$$X = x_1 x_2 \cdots x_m, \qquad Y = y_1 y_2 \cdots y_n \tag{2}$$

be chain-codes. A candidate window of Y is

$$W_{i,\ell} = y_i y_{i+1} \cdots y_{i+\ell-1}. \tag{3}$$

Its length ℓ is fixed or obtained from the selected window policy. Let

$$\mathcal{C}(Y) = \{(i, \ell) : W_{i,\ell} \text{ is admissible}\} \tag{4}$$

be the resulting candidate set and

$$z_{i,\ell} = \mathcal{N}(W_{i,\ell}) \tag{5}$$

[1] In this work, the alphabet is directional in the sense that its symbols encode quantized contour directions rather than ordinary characters. Specifically, each contour is represented as a chaincode—a sequence of symbols describing successive discrete step directions along the contour—over a finite windrose alphabet Σ, where each symbol corresponds to one quantized contour direction. The same abstraction applies whenever an object is encoded as a sequence of discrete symbols and compared under an edit-based distance.

the application-normalised candidate. The exact output is

$$\mathcal{M}_{\mathcal{C}}(X, Y, \tau) = \{(i, \ell) \in \mathcal{C}(Y) : \mathrm{Lev}(X, z_{i,\ell}) \leq \tau\}. \tag{6}$$

If $\mathcal{C}(Y)$ contains every factor of Y, this is the fixed-query substring component of semi-local comparison. The experiments instead use full-fragment, static, percentage-based, or cyclic subsets [21].

For contour matching, normalisation reverses traversal and replaces each direction by its opposite [5,19]:

$$\mathrm{opp}(\sigma_i) = \sigma_{(i+D/2) \bmod D}, \tag{7}$$

$$\mathrm{RO}(w_1 \cdots w_{|w|}) = \mathrm{opp}(w_{|w|}) \cdots \mathrm{opp}(w_1) \tag{8}$$

where $|w|$ denotes the number of symbols in w. For $1 \leq i \leq j \leq n$,

$$\mathrm{RO}(Y[i..j]) = \mathrm{RO}(Y)[n - j + 1..n - i + 1]. \tag{9}$$

Thus, candidate-wise normalisation is equivalent to extracting the mapped factor from one globally normalised chain-code. Direct approximate search of pattern X in $\mathrm{RO}(Y)$ is therefore possible. The restricted policies require intersection with the mapped set $\mathcal{C}(Y)$. Myers' method, for example, processes a text in $O(n\lceil m/w \rceil)$, where w is the machine-word width [14,9,26,8]. The present baseline instead applies standard $\Theta(m\ell)$ dynamic programming to each admitted candidate.

A filter is lossless when

$$\mathrm{LB}(X, z) \leq \mathrm{Lev}(X, z), \tag{10}$$

because

$$\mathrm{LB}(X, z) > \tau \implies \mathrm{Lev}(X, z) > \tau. \tag{11}$$

Only this implication is used for rejection [12]. Surviving candidates remain subject to exact verification.

3.2 Proposed Method

For each z, the cascade evaluates bounds $\mathrm{LB}_1, \ldots, \mathrm{LB}_t$ and uses

$$\mathrm{LB}_{\max}(X, z) = \max_{1 \leq j \leq t} \mathrm{LB}_j(X, z). \tag{12}$$

The candidate is rejected when $\mathrm{LB}_{\max} > \tau$, otherwise $\mathrm{Lev}(X, z)$ is computed. Hence, the cascade returns exactly the exhaustive accepted subset of the same $\mathcal{C}(Y)$.

The cascade used in this paper contains four base bounds:
- the length-difference bound $\mathrm{LB}_{\mathrm{len}}$,
- histogram bound $\mathrm{LB}_{\mathrm{hist}}$,
- transition bound $\mathrm{LB}_{\mathrm{trans}}$,
- contiguous g-gram bound LB_g.

The length bound is

$$\mathrm{LB}_{\mathrm{len}}(X, z) = \big||X| - |z|\big|, \tag{13}$$

because changing length requires insertions or deletions [11].

For $a \in \Sigma$, let $H_a(w)$ count occurrences of a in w. The histogram bound, formally defined in the literature as the bag distance[2] [3], is

$$\mathrm{LB}_{\mathrm{hist}}(X, z) = \left\lceil \frac{\|H(X) - H(z)\|_1}{2} \right\rceil. \tag{14}$$

One substitution changes at most two count components, and one insertion or deletion changes one.

For symbols σ_i, σ_j, define the signed turn, with a fixed half-turn tie convention, by

$$\Delta(\sigma_i, \sigma_j) = \Big((j - i + D/2) \bmod D\Big) - D/2. \tag{15}$$

Let $T(w)$ count adjacent turns. A substitution changes at most two turns and therefore changes the turn histogram by L_1-distance at most four. Insertion or deletion changes it by at most three. Consequently,

$$\mathrm{LB}_{\mathrm{trans}}(X, z) = \left\lceil \frac{\|T(X) - T(z)\|_1}{4} \right\rceil \tag{16}$$

is safe for the stated representation.

Let $G_g(w)$ be the multiset of contiguous g-grams and

$$c_g(X, z) = |G_g(X) \cap G_g(z)| \tag{17}$$

their multiplicity-aware intersection. The missing query grams are

$$M_g(X, z) = |G_g(X)| - c_g(X, z). \tag{18}$$

As one edit destroys at most g query grams,

$$\mathrm{LB}_g(X, z) = \left\lceil \frac{M_g(X, z)}{g} \right\rceil, \tag{19}$$

and every valid candidate must satisfy

$$c_g(X, z) \geq |G_g(X)| - \tau g. \tag{20}$$

The non-negative right-hand side

$$s_g = \max\{0, |G_g(X)| - \tau g\} \tag{21}$$

is termed the retained g-gram signature length. It becomes non-selective when $s_g = 0$. Small g is robust but less selective, while large g is more discriminative but makes Equation (20) inactive sooner as τ grows.

Table 1 illustrates safe rejection and filter-stage false positives.

For N candidates of lengths $\ell_1, \ldots, \ell_N$, candidate-wise dynamic programming costs

$$T_{\mathrm{base}} = \Theta\left(m \sum_{r=1}^{N} \ell_r\right), \tag{22}$$

[2] In this paper, the statement "histogram bound" refers only to "bag-distance bound" LB_{hist}, not to "turn histogram".

Table 1. Lower-bound outcomes for $X =$ `AABBCCDEEE`, $\tau = 2$, and $g = 2$.

Case	Candidate z	Main bound result	Exact d	Interpretation
1	`AABBCC`	$\mathrm{LB}_{\mathrm{len}} = 4 > \tau$	4	Rejected by length.
2	`AAAAABBBBB`	$\mathrm{LB}_{\mathrm{hist}} = 6 > \tau$	8	Rejected by histogram.
3	`EACBDAECBD`	$\mathrm{LB}_{\mathrm{hist}} = 1,\ \mathrm{LB}_{\mathrm{trans}} = 5 > \tau$	8	Rejected by turns despite plausible symbol counts.
4	`ABCEEBCADD`	$\mathrm{LB}_{\mathrm{trans}} = 2,\ \mathrm{LB}_g = 3 > \tau$	7	Rejected by insufficient common bigrams.
5	`BCDEECCEDA`	$\max_i \mathrm{LB}_i = 2 \le \tau$	8	Survives filtering, rejected exactly.
6	`AABBCCDEED`	$\max_i \mathrm{LB}_i = 1 \le \tau$	1	Survives filtering and is accepted.

or $\Theta(Nm\ell)$ for approximately equal ℓ. If C_{LB} is the mean cascade cost and p is the survival rate,

$$T_{\mathrm{cascade}} \approx NC_{\mathrm{LB}} + pNm\ell. \tag{23}$$

Filtering is beneficial relative to this baseline when

$$p < 1 - \frac{C_{\mathrm{LB}}}{m\ell}. \tag{24}$$

This model does not compare the cascade with whole-text bit-parallel or semi-global algorithms [14,26,8]. Length requires constant auxiliary space. Symbol and turn histograms require space proportional to their alphabets. Explicit g-gram profiles are linear in chain-code length, and rolling-row verification uses $\Theta(\min\{m, \ell\})$ space.

4 Experimental Evaluation

The C#/.NET 10 implementation used no external algorithmic libraries. PowerShell scripts generated experiment matrices, warm-ups, repeated timings, rejected-candidate audits, and CSV summaries. Tests ran under Windows 10 on a Surface Pro 7 Plus with an Intel i7-1165G7 2.80 GHz CPU and 16 GB RAM. Exact verification used a custom rolling-row Levenshtein routine.

The 635 configurations comprised 268 generated, 304 exhaustive, and 63 Afrosiyab profiles. Generated data used either independent uniform symbols or a geometry-aware generator that rejected immediate opposite steps and self-intersecting traces. The central equal-length setting used $D = 8$, $|X| = |z| = 10$, $\tau = 2$, $g = 2$, ten seeds, and independently resampled measured repeats. Variable-length candidates used $|X| = 12$ and $|z| \in [7, 13]$ with $\tau \in \{2, 3, 4\}$. Additional sweeps varied $D \in \{4, 6, 8, 12, 16\}$, equal lengths in $\{6, 8, 10, 12, 14, 18\}$, $\tau \in \{1, \ldots, 5\}$, $g \in \{1, \ldots, 10\}$, active bounds, and candidate counts from 10^5 to 10^6.

Exhaustive validation enumerated the equal-length spaces

$$(|\Sigma|, |X|, |Y|) \in \{(2, 5, 5), (2, 6, 6), (4, 3, 3), (4, 4, 4)\}$$

and the variable-length spaces

$$(|\Sigma|, |X|, |Y|) \in \{(2, 5, 3), (2, 6, 4), (4, 4, 3)\}.$$

Equal-length cases used full chain-codes, cyclic static windows, and cyclic windows of 25% and 50%. Variable-length cases used full chain-codes. Thresholds were 0–3 and 1–4, respectively. These runs validated correctness rather than speed.

The real dataset [19] contains 17 Afrosiyab fragment chain-codes over $D = 16$, of lengths $49, 42, 46, 49, 47, 38, 43, 43, 34, 61, 53, 53, 63, 46, 45, 41, 45$. Their minimum, median, mean, and maximum are 34, 46, 46.94, and 63. Reverse-opposite normalisation was enabled for real comparisons and disabled for generated and exhaustive data.

Real workloads retained one direction for each unordered fragment pair. Full-fragment tests, therefore, used $\binom{17}{2} = 136$ pairs. For static non-cyclic length cw, define

$$q_{cw}(F) = \max\{0, |F| - cw + 1\}, \tag{25}$$

$$N_{\text{all}}(cw) = \sum_{i<j} q_{cw}(F_i) q_{cw}(F_j). \tag{26}$$

For $cw \in \{8, 10, 12, 14, 16, 20\}$, workloads were 216 518, 195 334, 175 238, 156 230, 138 310, and 105 734, respectively. Dynamic pairwise length was

$$cw_{ij} = \text{round}(p \min\{|F_i|, |F_j|\}), \tag{27}$$

with midpoint values rounded away from zero. Percentages $p \in \{0.15, 0.20, 0.25, 0.30\}$ produced length ranges 5–9, 7–12, 9–15, and 10–18, and workloads 234 268, 209 008, 185 466, and 165 253. Main window sweeps used $\tau = 5$ and $g = 2$. Additional runs varied g, active bounds, and τ for fragments F_{10} and F_{11}.

Because LB_g is query-asymmetric, unordered window pairs are evaluated as

$$\text{LB}_g(x, y) = \max\{\text{LB}_g(x \to y), \text{LB}_g(y \to x)\}. \tag{28}$$

The maximum of two valid lower bounds remains valid, but it doubles the directional evaluation cost of this bound.

Table 2 summarises the actual verifier inputs.

Table 2. Lengths and thresholds used in the experiment matrix.

Family	Comparison policy	$m = \lvert X \rvert$	$\ell = \lvert z \rvert$	τ
Exhaustive	Equal-length full chain-codes	$\{3, 4, 5, 6\}$	$\{3, 4, 5, 6\}$	0–3
Exhaustive	Equal-source cyclic static windows	$\{3, 4\}$	$\{3, 4\}$	0–3
Exhaustive	Equal-source cyclic 25% and 50% windows	1–3	1–3	0–3
Exhaustive	Variable-length full chain-codes	$\{4, 5, 6\}$	$\{3, 4\}$	1–4
Generated	Equal-length chain-codes	$\{6, 8, 10, 12, 14, 18\}$	$\{6, 8, 10, 12, 14, 18\}$	1–5
Generated	Variable-length candidates	12	7–13	2–4
Real Afrosiyab	Full-fragment comparisons	34–63	34–63	4–6
Real Afrosiyab	Non-cyclic static windows	$\{8, 10, 12, 14, 16, 20\}$	$\{8, 10, 12, 14, 16, 20\}$	5
Real Afrosiyab	Non-cyclic dynamic windows, 15%–30%	5–18	5–18	5
Focused F_{10}–F_{11}	Static $cw = 14$ / dynamic 25%	14/13	14/13	4–8

Most patterns fit within one 64-bit word, so single-word/chain-code bit-parallel Shift–Or, approximate BNDM, and Myers-style methods are applicable in principle [15,10,14]. Applicability does not establish relative performance: it also depends on τ/m, alphabet size, output requirements, and implementation. Reported speedups are therefore relative only to the stated candidate-wise dynamic-programming baseline, not to Shift–Or, approximate BNDM, Myers, or Edlib [14,26].

Each profile compared exhaustive exact verification over the same candidate set with the cascade followed by exact verification. Warm-ups were excluded, measured

repeats were aggregated, and rejected candidates were audited against exact distance. Let N_{verify} be the number surviving the cascade. The principal metrics were

$$p = \frac{N_{\text{verify}}}{N_{\text{all}}}, \qquad r = 1 - p, \tag{29}$$

and

$$\text{speedup} = \frac{T_{\text{exhaustive}}}{T_{\text{filter+verify}}}. \tag{30}$$

Ground-truth and filter-and-verify matches are denoted gt_{matches} and fv_{matches}, respectively. False negatives equal $gt_{\text{matches}} - fv_{\text{matches}}$. Timing tables report means and 95% confidence intervals across repeated runs. Logs also contain sequential and isolated bound counts and timings for ablation.

5 Results

No false negatives occurred in any exhaustive, generated, real-data, timing, or audit result (Table 3). The empirical pipeline was therefore consistent with exhaustive verification across the tested matrix.

Table 3. Correctness evidence across validation sources.

Evidence source	Rows/configs.	N_{all}	gt matches	fv matches	Recall
Exhaustive validation	304	61 886 464	38 626 536	38 626 536	1.000
Generated summary	268	63 400 000	20 837	20 837	1.000
Real Afrosiyab summary	63	7 611 060	114 043	114 043	1.000
Measured timing runs	780	76 240 710	1 145 020	1 145 020	1.000

Generated data showed strong pruning under selective settings (Table 4). At $cw = 10$, $\tau = 2$, and $g = 2$, survival was 0.160% for uniform and 0.356% for geometry-aware data, with 4.05× and 3.81× mean speedups. Variable-length tests at $\tau = 2$ reduced survival below 0.025% and exceeded 7× speedup. Both benefits weakened as τ increased.

Table 4. Representative generated configurations using the full cascade.

Configuration	N_{all}	Survival	Mean speedup
Uniform equal, $cw = 10$, $\tau = 2$, $g = 2$	300 000	0.160%	4.05×
Geometry equal, $cw = 10$, $\tau = 2$, $g = 2$	300 000	0.356%	3.81×
Uniform variable, $\tau = 2$	150 000	0.005%	7.80×
Uniform variable, $\tau = 3$	150 000	0.440%	4.96×
Uniform variable, $\tau = 4$	150 000	9.015%	2.25×
Geometry variable, $\tau = 2$	150 000	0.024%	7.44×
Geometry variable, $\tau = 3$	150 000	0.865%	4.98×
Geometry variable, $\tau = 4$	150 000	9.913%	2.41×

Afrosiyab results were more sensitive to window choice (Table 5). Dynamic survival fell from 41.573% at 15% to 6.469% at 30%. Speedup crossed parity at 20% and reached 3.19× at 30%. Static survival fell from 24.150% at $cw = 8$ to 0.316% at $cw = 20$. The $cw = 16$ setting retained 35 accepted matches and achieved 5.46×, whereas $cw = 20$ accepted none.

Table 5. Afrosiyab results for non-cyclic windows, the full cascade, $au = 5$, and $g = 2$.

Setting	N_{all}	N_{verify}	Survival	Reduction	Matches	False positives	Mean speedup	95% CI
Dynamic 15%	234 268	97 393	41.573%	58.427%	62 866	34 527	0.47×	[0.45, 0.50]
Dynamic 20%	209 008	43 053	20.599%	79.401%	9 195	33 858	1.06×	[1.02, 1.10]
Dynamic 25%	185 466	20 417	11.008%	88.992%	1 289	19 128	1.90×	[1.84, 1.96]
Dynamic 30%	165 253	10 690	6.469%	93.531%	307	10 383	3.19×	[2.92, 3.45]
Static $cw = 8$	216 518	52 289	24.150%	75.850%	13 279	39 010	0.81×	[0.77, 0.85]
Static $cw = 10$	195 334	27 325	13.989%	86.011%	2 473	24 852	1.48×	[1.40, 1.56]
Static $cw = 12$	175 238	13 792	7.870%	92.130%	391	13 401	2.53×	[2.37, 2.69]
Static $cw = 14$	156 230	6 497	4.159%	95.841%	85	6 412	3.86×	[3.62, 4.11]
Static $cw = 16$	138 310	2 738	1.980%	98.020%	35	2 703	5.46×	[5.14, 5.79]
Static $cw = 20$	105 734	334	0.316%	99.684%	0	334	10.11×	[9.27, 10.95]

This behavior is therefore attributable to the candidate-policy being overly restrictive rather than to a false negative in the filter.

The 95% timing intervals support stable conclusions for the main profiles: dynamic 25% achieved 1.90× [1.84, 1.96], and static $cw = 14$ achieved 3.86× [3.62, 4.11]. Parameter sweeps further support H3. Increasing τ raised survival and eventually made filtering slower than exhaustive verification (Table 6).

Table 6. Threshold sweep for $cw = 10$ and $g = 2$.

Dataset	τ	Survival	Reduction	Mean speedup
Uniform	1	0.001%	99.999%	4.84×
Uniform	2	0.162%	99.838%	3.99×
Uniform	3	5.832%	94.168%	2.01×
Uniform	4	43.315%	56.685%	0.93×
Uniform	5	78.694%	21.306%	0.60×
Geometry	1	0.002%	99.998%	4.54×
Geometry	2	0.250%	99.750%	4.01×
Geometry	3	5.906%	94.094%	2.06×
Geometry	4	34.188%	65.812%	1.15×
Geometry	5	65.092%	34.908%	0.72×

For fixed $\tau = 2$ and $g = 2$, increasing cw reduced survival (Table 7). This is consistent with Equation (20): $|G_g(X)|$, and hence s_g, grows with $m = cw$ when g and τ are fixed. Real contours nevertheless impose an application constraint: excessively long windows may exclude short physical correspondences before filtering.

Increasing D also reduced accidental agreement (Table 8). Speedup peaked before the smallest survival values because, once exact verification is nearly eliminated, bound computation dominates.

Runtime scaled approximately linearly with candidate count (Table 9). Throughput remained near 3 200 candidates/ms. Geometry aware data showed the same pattern – near 3 000 candidates/ms. These tests establish scalability in N_{all}, not independently in physical contour length.

To summarise the practical trade-off between pruning strength and runtime benefit, Figure 2 plots real-data configurations by survival rate and mean speedup.

The figure complements the tabular results by showing that candidate reduction alone is not sufficient to evaluate a filter. A configuration is practically useful only

Table 7. Window-length sweep for $\tau = 2$ and $g = 2$.

Dataset	cw	Survival	Mean speedup
Uniform	6	12.759%	1.18×
Uniform	8	1.562%	2.47×
Uniform	10	0.162%	3.94×
Uniform	12	0.016%	5.00×
Uniform	14	0.002%	7.02×
Uniform	18	0.000%	11.43×
Geometry	6	10.594%	1.26×
Geometry	8	1.808%	2.31×
Geometry	10	0.250%	4.11×
Geometry	12	0.051%	5.68×
Geometry	14	0.010%	7.45×
Geometry	18	0.001%	8.61×

Table 8. Alphabet-size sweep for $cw = 10$, $\tau = 2$, and $g = 2$.

Dataset	Alphabet size D	Survival	Mean speedup
Uniform	4	16.904%	1.79×
Uniform	6	1.442%	3.10×
Uniform	8	0.162%	3.80×
Uniform	12	0.006%	3.70×
Uniform	16	0.001%	3.60×
Geometry	4	16.209%	1.92×
Geometry	6	2.508%	2.97×
Geometry	8	0.250%	3.97×
Geometry	12	0.010%	3.60×
Geometry	16	0.001%	3.46×

Table 9. Scale tests over eight runs.

Dataset	N_{all}	Filtered time (ms)	Exhaustive time (ms)	Speedup	Throughput (cand/ms)
Uniform	100 000	30.49	126.17	4.14×	3,280
Uniform	500 000	153.58	622.59	4.07×	3,256
Uniform	1 000 000	296.12	1 200.05	4.05×	3,377
Geometry	100 000	32.18	126.09	3.92×	3,107
Geometry	500 000	173.16	681.35	3.94×	2,888
Geometry	1 000 000	344.59	1 348.09	3.92×	2,902

when the reduction in exact Levenshtein verification outweighs the overhead of computing the lower bounds. Marker size indicates the number of accepted matches, which helps distinguish useful pruning from over-restrictive configurations that obtain high speedup but return few or no matches.

Dynamic 15% pruned candidates but was slower than the baseline. Static $cw = 20$ was fast but accepted no match. Intermediate settings combined non-zero output with net acceleration. Ablation identifies the source of this trade-off. The dynamic 25% and static $cw = 14$ settings were selected as representative non-extreme operating points for the adaptive and fixed-window policies, respectively, because both achieved substantial pruning while retaining non-zero real matches. For dynamic 25% and static $cw = 14$, histogram-only filtering nearly matched the full cascade's survival but

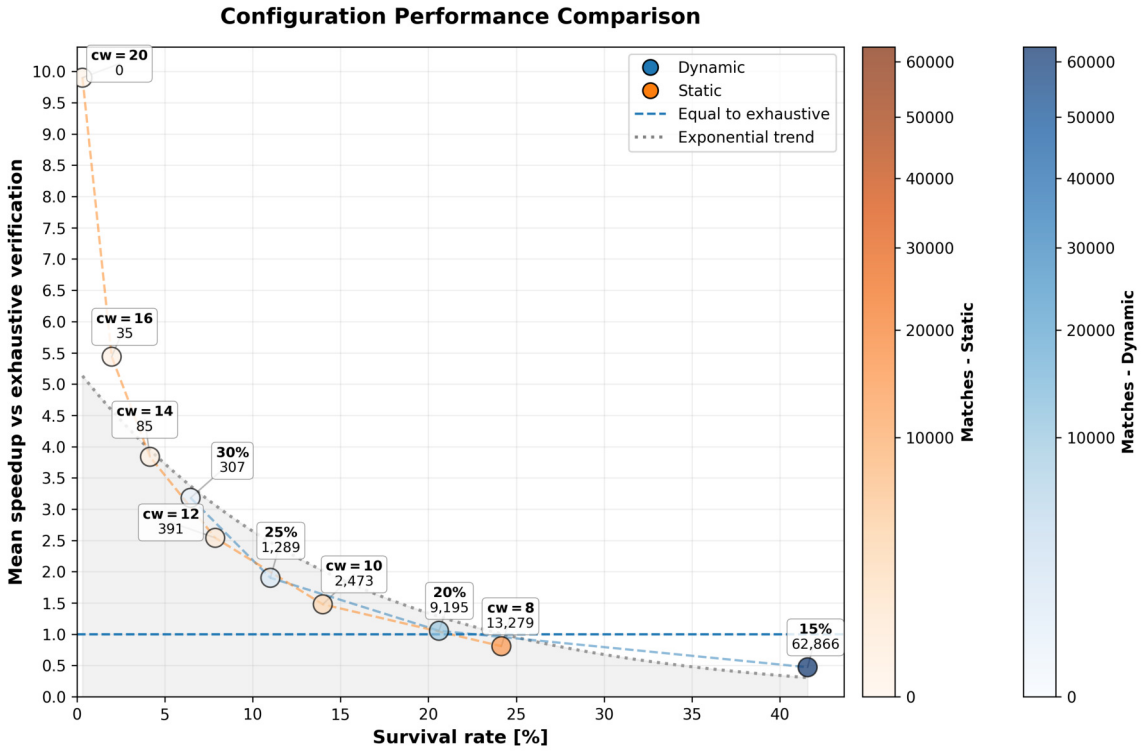

Figure 2. Afrosiyab survival–speedup trade-off. The dashed line marks parity with exhaustive verification. Marker size represents accepted matches.

was faster (Figure 3). $\mathrm{LB}_{\mathrm{len}}$ was inactive for equal-length windows, $\mathrm{LB}_{\mathrm{trans}}$ contributed almost no pruning (Figure 4), and symmetric g-gram evaluation LB_g cost more than it saved in these real profiles.

6 Discussion

H1 follows from the bounds in Section 3. Table 3 corroborates their implementation. H2 is supported as reduced candidate-wise verification, not as an unconditional speedup: rejected work must exceed the filtering overhead. H3 is supported by the effects of τ, cw, D, s_g, data type, and active bounds.

A safe bound need not be useful in every regime. Length is inactive for equal windows. Turns were too coarse, and symmetric g-grams were too costly in the selected Afrosiyab profiles. Scheduling should therefore prioritise cheap, selective bounds.

The chief empirical limitation is the reliance on a single real dataset. Although this case originates from a real cultural heritage reconstruction problem and exhibits realistic contour properties, additional broken-object datasets are required before broader empirical generality can be claimed. The synthetic experiments provide controlled evidence under varied parameters, but they cannot fully replace evaluation on heterogeneous real fragment collections.

Timings are hardware and implementation dependent, and the baseline is candidate-wise dynamic programming rather than direct semi-local, bit-parallel, SIMD, or indexed search. The general conclusions are thus losslessness and verification reduction relative to that baseline.

Figure 3. Pairwise Afrosiyab ablation differences in mean speedup. Values are row minus column: positive speedup differences favour the row. L, H, T, G, A denote $\mathrm{LB}_{\mathrm{len}}$, $\mathrm{LB}_{\mathrm{hist}}$, $\mathrm{LB}_{\mathrm{trans}}$, LB_{g}, and $\mathrm{LB}_{\mathrm{all}}$.

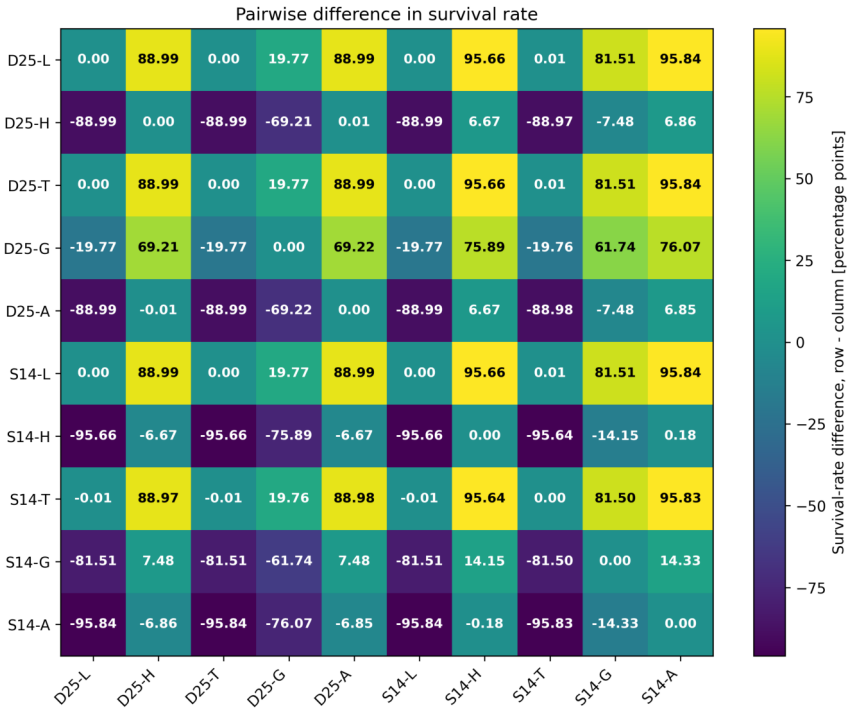

Figure 4. Pairwise Afrosiyab ablation differences in survival rate. Values are row minus column: negative survival differences indicate stronger row pruning. L, H, T, G, A denote $\mathrm{LB}_{\mathrm{len}}$, $\mathrm{LB}_{\mathrm{hist}}$, $\mathrm{LB}_{\mathrm{trans}}$, LB_{g}, and $\mathrm{LB}_{\mathrm{all}}$.

7 Conclusion and Future Work

The cascade safely rejects contour candidates on the basis of length, symbol composition, signed turns, and g-gram overlap before exact Levenshtein verification. Across 635 configurations, it matched exhaustive output and substantially reduced the number of exact checks in selective regimes. Runtime and practical value nevertheless depended on window policy, threshold, alphabet, and bound scheduling. Filtering preserves only those matches represented by the chosen candidate set.

Future work should:

- add more diverse real datasets,
- improve the selection of cw, τ, g, s_g, D,
- optimise ordering, caching, and indexing,
- compare the method against equivalent-output whole-text bit-parallel and semi-local baselines, and
- adapt BLAST-like and FASTA-style seeds, diagonals, scores, and extensions to chaincodes, then measure runtime, recall, and false-negative rate against exhaustive ground truth.

Declaration of Generative AI in the Writing Process

The author acknowledges the use of AI tools during the research process and paper writing, specifically: Perplexity (during the initial drafting stage of this manuscript), ChatGPT (during the experiment planning phase), Copilot (during the implementation phase and for paper formatting), and NotebookLM (for literature synthesis and argument outlining). All AI-generated suggestions were critically evaluated and verified against primary literature. The author retains full accountability for the accuracy and originality of the final manuscript.

References

1. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman: *Basic local alignment search tool.* Journal of Molecular Biology, 215(3) 1990, pp. 403–410.
2. S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman: *Gapped BLAST and PSI-BLAST: A new generation of protein database search programs.* Nucleic Acids Research, 25(17) 1997, pp. 3389–3402.
3. I. Bartolini, P. Ciaccia, and M. Patella: *String matching with metric trees using an approximate distance*, in String Processing and Information Retrieval, A. H. F. Laender and A. L. Oliveira, eds., Berlin, Heidelberg, 2002, Springer Berlin Heidelberg, pp. 271–283.
4. Z. Bingöl, M. Alser, O. Mutlu, O. Ozturk, and C. Alkan: *Gatekeeper-gpu: Fast and accurate pre-alignment filtering in short read mapping.* IEEE Transactions on Computers, 73(5) 2024, pp. 1206–1218.
5. E. Bribiesca, F. Bribiesca-Contreras, Ángel Carrillo-Bermejo, G. Bribiesca-Correa, and N. Hevia-Montiel: *A chain code for representing high definition contour shapes.* Journal of Visual Communication and Image Representation, 61 2019, pp. 93–104.
6. S. Burkhardt: *Filter algorithms for approximate string matching*, doctoral dissertation, University of Saarland, 2002.
7. S. Burkhardt and J. Kärkkäinen: *Better filtering with gapped q-grams*, in Combinatorial Pattern Matching, A. Amir, ed., Berlin, Heidelberg, 2001, Springer Berlin Heidelberg, pp. 73–85.
8. P. Charalampopoulos, T. Kociumaka, and P. Wellnitz: *Faster pattern matching under edit distance : A reduction to dynamic puzzle matching and the seaweed monoid of permutation matrices*, in 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS), 2022, pp. 698–707.

9. J. Daily: *Parasail: Simd c library for global, semi-global, and local pairwise sequence alignments.* BMC Bioinformatics, 17(1) Feb 2016, p. 81.
10. S. Faro and T. Lecroq: *Twenty years of bit-parallelism in string matching*, in Festschrift for Bořivoj Melichar, J. Holub, B. W. Watson, and J. Žďárek, eds., Prague Stringology Club, 2012, pp. 72–101.
11. M. Khalid, M. M. Yousaf, and M. U. Sadiq: *Toward efficient similarity search under edit distance on hybrid architectures.* Information, 13(10) 2022, p. 452.
12. Y. Liang and P. Zhao: *Similarity search in graph databases: A multi-layered indexing approach*, in 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 783–794.
13. Y. Liu, H. Wei, and H. Cheng: *Exploiting lower bounds to accelerate approximate nearest neighbor search on high-dimensional data.* Information Sciences, 465 2018, pp. 484–504.
14. G. Myers: *A fast bit-vector algorithm for approximate string matching based on dynamic programming.* J. ACM, 46(3) May 1999, p. 395–415.
15. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, 2002.
16. W. R. Pearson and D. J. Lipman: *Improved tools for biological sequence comparison.* Proceedings of the National Academy of Sciences of the United States of America, 85(8) 1988, pp. 2444–2448.
17. L. Renders, L. Depuydt, S. Rahmann, and J. Fostier: *Automated design of efficient search schemes for lossless approximate pattern matching*, in Research in Computational Molecular Biology, J. Ma, ed., Cham, 2024, Springer Nature Switzerland, pp. 164–184.
18. L. Renders, L. Depuydt, S. Rahmann, and J. Fostier: *Lossless approximate pattern matching: Automated design of efficient search schemes.* Journal of Computational Biology, 31(10) Oct. 2024, p. 975–989.
19. S. P. Skulimowski and J. Montusiewicz: *A novel approach for assemblage of historical artefacts using the levenshtein distance and feedback loop.* Journal of Cultural Heritage, 75 2025, pp. 158–167.
20. M. Steinegger and J. Söding: *MMseqs2 enables sensitive protein sequence searching for the analysis of massive data sets.* Nature Biotechnology, 35(11) 2017, pp. 1026–1028.
21. A. Tiskin: *Semi-local string comparison: Algorithmic techniques and applications.* Mathematics in Computer Science, 1 2008, pp. 571–603.
22. E. Ukkonen: *Approximate string-matching with q-grams and maximal matches.* Theoretical Computer Science, 92(1) 1992, pp. 191–211.
23. C. Xiao, W. Wang, and X. Lin: *Ed-join: an efficient algorithm for similarity joins with edit distance constraints.* Proc. VLDB Endow., 1(1) Aug. 2008, p. 933–944.
24. A. Zeakis, D. Skoutas, D. Sacharidis, O. Papapetrou, and M. Koubarakis: *Tokenjoin: Efficient filtering for set similarity join with maximum weighted bipartite matching.* Proc. VLDB Endow., 16(4) Dec. 2022, p. 790–802.
25. X. Zhao, C. Xiao, X. Lin, W. Zhang, and Y. Wang: *Efficient structure similarity searches: a partition-based approach.* The VLDB Journal, 27(1) Feb 2018, pp. 53–78.
26. M. Šošić and M. Šikić: *Edlib: a C/C++ library for fast, exact sequence alignment using edit distance.* Bioinformatics, 33(9) 05 2017, pp. 1394–1395.

Computation of Periods and Covers Using Sampling

Thierry Lecroq[1], Arnaud Lefebvre[1], and Francesco Pio Marino[1,2]

[1] Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie Univ, LITIS UR 4108, Rouen, France
[2] Department of Mathematics and Computer Science, University of Catania, Catania, Italy

Abstract. Identifying regularities in strings, such as *periods* and *covers*, is crucial for applications in text compression, computational biology, and pattern recognition. *Characters-Distance-Sampling* (CDS) is an efficient technique that encodes a string by storing distances between selected pivot characters, accelerating string processing tasks. We apply CDS to compute periods and shortest covers, selecting only the first character as the pivot. This strategy yields optimized computations, achieving speedups of 81%–85% for period computation and 34%–40% for cover detection. These results demonstrate the potential of CDS-based representations for efficient string analysis and broader applications.

1 Introduction

String analysis is fundamental in various domains, including literature, linguistics, computer science, and computational biology, where DNA, RNA, and protein sequences are represented as character chains. Identifying structural regularities in strings is essential for efficient data processing, pattern recognition, and compression. These notions are widely explored in combinatorial pattern matching and have applications in text indexing, bioinformatics, and data compression. Among these, *periods* [3] and *quasiperiods* (*covers*) [1,13] are widely studied in combinatorial pattern matching, with applications in text indexing, bioinformatics, and data compression.

In recent years, *Characters-Distance-Sampling* (CDS) has been successfully applied in string processing tasks such as exact pattern matching [6,10,11]. CDS represents a string by storing the distances between occurrences of selected *pivot characters*, typically leading to a more compact representation of the string [9]. However, despite its effectiveness in searching and indexing, CDS has not yet been leveraged for computing string regularities.

In this paper, we present the first application of CDS to the computation of fundamental string regularities, specifically the *period* and the *shortest cover* of a string. We develop efficient algorithms that operate directly on the CDS representation of a string, significantly improving computational efficiency compared to classical methods. Our experimental results demonstrate that these methods achieve substantial speedups over traditional approaches.

The remainder of this paper is organized as follows. Section 2 introduces fundamental concepts and notations. Section 3 presents our method for computing the period of a string using its CDS representation. In Section 4, we describe how to compute the shortest cover of a string using the same representation. Experimental results are detailed in Section 5, followed by our conclusions and future research directions in Section 6.

Thierry Lecroq, Arnaud Lefebvre, Francesco Pio Marino: Computation of Periods and Covers Using Sampling, pp. 85–96.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

2 Preliminaries

A *string* is a sequence of zero or more characters drawn from a finite *alphabet* Σ, with $\sigma = |\Sigma|$ denoting the number of distinct characters. The empty string, containing zero characters, is denoted by ε, and the set of all finite strings over Σ is Σ^*. A string x of length m is represented as $x[0\,..\,m-1]$, where $x[i] \in \Sigma$ for $0 \leq i \leq m-1$, and where $|x| = m$ denotes the length of the string. A string u is a *prefix* of x if $x = uw$ for some $w \in \Sigma^*$, and a *suffix* if $x = wu$. If u is both a prefix and a suffix of x but $u \neq x$, it is called a *border*. By definition, the empty string is a border of any non-empty string. A string may have multiple borders, with the longest one referred to as *the border* of x, denoted by $Border(x)$.

An integer p $(1 \leq p \leq |x|)$ is a *period* of x if $x[i] = x[i+p]$ for all $0 \leq i < |x| - p$. The smallest such p is called *the period* of x, denoted by $per(x)$. It is well known that x has period p if and only if it has a border of length $|x| - p$ [14]. Specifically, for a non-empty string x: $|x| = per(x) + |Border(x)|$. A string x is *periodic* if $per(x) \leq |x|/2$, it is *non-periodic* otherwise. Borders and periods are dual concepts since each period corresponds to a border, and vice versa. All borders of a non-empty string x can be found recursively through the borders of its longest border. In online string matching, a string's period determines the optimal shift length when an occurrence is found.

A related concept is that of *covers*. A string w of length ℓ is a *cover* (or *quasiperiod*) of a string x of length m if every position in x is included in some occurrence of w. This ensures that every character of x is contained within at least one occurrence of w, though occurrences may overlap. A string can have multiple covers, with the shortest among them called the *shortest cover*. If a string x has no proper cover (i.e., its only cover is itself), it is called *superprimitive*; otherwise, it is *quasiperiodic*.

Let $pos(u, x)$ be the ordered list of all starting positions where u appears in x: $pos(u, x) = (i \mid x[i\,..\,i + |u| - 1] = u)$. Then $\{0, |x| - |w|\} \subseteq pos(w, x)$ if w is a cover of x, meaning that a cover is a border.

2.1 Characters-Distance-Sampling in Brief

We now provide a concise overview of the methodology used to construct a partial index in the `CDS` setting. To build the partial index, a subset $C \subseteq \Sigma$ is chosen as the *set of pivot characters*. The input string x of length m is then sampled by computing the distances between consecutive occurrences of any pivot character $c \in C$. For the remaining of the paper we refer at a single pivot character c, we notice that all the subsequent methods can be extended to a non-empty subset of pivot character C of any size.

Formally, given the function $\delta : \{0, \ldots, \dot{m} - 1\} \to \{0, \ldots, m - 1\}$, where $\delta(i)$ is the position of the $(i+1)$-th occurrence of a pivot character c in x, we first obtain the *position-sampling* version of x, indicated by $\dot{x}$, as the numeric sequence of length $\dot{m}$, defined as $\dot{x} = \langle \delta(0), \delta(1), \ldots, \delta(\dot{m} - 1) \rangle$ where $x[\delta(i)] = c$ for $0 \leq i < \dot{m}$ and $x[j] \neq c$ for $\delta(i) < j < \delta(i+1)$ for $0 \leq i < \dot{m} - 1$ and $\delta(\dot{m} - 1) < j < m - 1$. Then, given the function Δ, where $\Delta(i) = \delta(i+1) - \delta(i)$ for $0 \leq i < \dot{m} - 1$, is the distance between two consecutive occurrences of any pivot character in x, we get the *character-distance-sampling* version of the text x, indicated by $\bar{x}$, as the numeric sequence of length $\bar{m} = \dot{m} - 1$, defined as $\bar{x} = \langle \Delta(0), \Delta(1), \ldots, \Delta(\bar{m} - 1) \rangle = \langle \delta(1) - \delta(0), \delta(2) - \delta(1), \ldots, \delta(\dot{m} - 1) - \delta(\dot{m} - 2) \rangle$.

For example, let $x =$ `agaacgcagtata` be a DNA sequence of length 13, over the alphabet $\Sigma =$ {`a`, `c`, `g`, `t`}. Let $C =$ {`a`} be the set of pivot characters. Thus,

the position-sampling version of x is $\dot{x} = \langle 0, 2, 3, 7, 10, 12\rangle$. The character-distance-sampling version of x is $\bar{x} = \langle 2, 1, 4, 3, 2\rangle$. Specifically, $\bar{x}[0] = \Delta(0) = \delta(1) - \delta(0) = 2 - 0 = 2$, while $\bar{x}[2] = \Delta(2) = \delta(3) - \delta(2) = 7 - 3 = 4$, and so on.

When searching for a pattern p, in a larger text t, a preprocessing step computes their character-distance-sampling versions $\bar{p}$ and $\bar{t}$, after which any string matching algorithm can be applied to find occurrences of $\bar{p}$ in $\bar{t}$ (see [9] for details). In its original design, the algorithm maintains the position-sampling version $\dot{t}$ and dynamically computes $\bar{t}$ during the search. This approach balances memory usage, as storing distances or positions requires the same space, but retaining positions allows direct access to the original text for verification.

However, in this formulation, each element of the sampled text requires 4 bytes of memory. An alternative implementation [8] reduces memory usage by shifting from position-based to distance-based representation, introducing the concept of *fake samples*. Here, distances exceeding a fixed threshold are decomposed into a sequence of smaller distances with inserted fake samples. This method reduces memory consumption, allowing representation with just 8 bits per element (for a threshold of 256).

Experimental results indicate that the additional space required to store the sampled text ranges from 2.5% to 20%, depending on the pattern length [8]. Furthermore, recent studies demonstrate that the Characters-Distance-Sampling algorithm enables any exact string matching procedure to be used as the search method [7]. Notably, search times are reduced by up to 96% compared to applying the same procedure to the non-sampled version of the text.

Additionally, new bounds and optimal sampling strategies have recently been studied by Faro *et al.* [5].

2.2 Classical Computation of the Period of a String

A fundamental approach for computing the period of a string x relies on determining the length of the border of all its prefixes. The key observation is that the length of the longest border of x directly determines its period.

The *border array* of a non-empty string x of length m is an array of size $m + 1$ defined as follows:

$$border_x[0] = -1, \quad border_x[i] = |Border(x[0\,..\,i-1])| \quad \text{for } 1 \le i \le |x|.$$

Each entry $border_x[i]$ represents the length of the longest border for the prefix of length i of x, namely $x[0\,..\,i-1]$. This array can be computed in linear time using the classical algorithm presented in [3]. This structure allows the period of the string to be determined efficiently using the relation $per(x) = m - border_x[m]$, where m is the length of x. The pseudo-code for the construction of the border array is reported in Figure 1.

Figure 2 illustrates the computation of the border array for the string $x =$ `abaababaaba`. The border array $border_x$ encodes structural information about the string by tracking the longest proper border of each prefix of x.

For instance, the prefix $x[0\,..\,4] =$ `abaab` has a border of length 2, meaning that its first two characters match its last two characters. Similarly, the prefix $x[0\,..\,10] =$ `abaababaaba` has a border of length 6, indicating that the first six characters match the last six.

For the full string of length $m = 11$, the border array value at index m is given by $border_x[11] = 6$. Since the period of a string is calculated as $m - border_x[m]$, we

```
BORDERS(x, m)
1  border_x[0] ← −1
2  i ← 0
3  for j ← 1 to m − 1 do
4     border_x[j] ← i
5     while i > −1 and x[i] ≠ x[j] do
6        i ← border_x[i]
7     i ← i + 1
8  border_x[m] ← i
9  return border_x
```

Figure 1. Classical computation of the border array of string x of length m.

i	0	1	2	3	4	5	6	7	8	9	10	11
$x[i]$	a	b	a	a	b	a	b	a	a	b	a	
$border_x[i]$	-1	0	0	1	1	2	3	2	3	4	5	6

Figure 2. Computed border array of $x =$ abaababaaba.

derive: $per(x) = 11 - border_x[11] = 11 - 6 = 5$. This result confirms that x repeats itself every 5 characters.

2.3 Classical Computation of the Shortest Cover of a String

Since a cover is inherently a border, computing the shortest cover of a string requires first identifying all its non-empty borders. For a string x of length m, let us define the sequence of borders recursively as follows:

$$Border^0(x) = x, \quad Border^i(x) = Border(Border^{i-1}(x)) \quad \text{for } i > 0.$$

The sequence of non-empty borders is then:

$$Border^1(x), Border^2(x), \ldots, Border^k(x) \neq \varepsilon, \quad \text{such that } Border^{k+1}(x) = \varepsilon.$$

The shortest cover of x is the smallest non-empty border $Border^j(x)$ that covers all the borders $Border^g(x)$ for $0 \leq g < j$. Explicit checks need to be performed to verify that a short border covers a longer border.

The algorithm starts by computing the shortest non-empty border and iteratively checks whether each border covers longer ones [2]. If a border $Border^j(x)$ fails to cover some $Border^h(x)$ with $h < j$, the process resumes with $j = h$. To ensure a linear-time complexity, only non-periodic borders need to be considered.

From Figure 2, we analyze how the shortest cover of the string $x =$ abaababaaba is determined. A cover of a string is a repeating pattern that occurs frequently enough to cover every position in the string.

For instance, to compute the shortest cover of a string x, we first extract all the non-empty borders of x. For instance, the decreasing sequence of the non-empty border lengths for $x[0\,..\,10] =$ abaababaaba is:

$$(border_x^1[|x|] = 6, border_x^2[|x|] = 3, border_x^3[|x|] = 1).$$

This means that the sequence of borders in increasing length of x is: (a, aba, abaaba). However, abaaba is periodic and does not need to be considered for computing a cover of x since it is actually covered by its border aba.

```
ShortestCover(x, m)
 1  border_x ← Borders(x, m)
 2  S ← EmptyStack()
 3  Push(S, m)
 4  i ← border_x[m]
 5  while i > 0 do
 6     if border_x[i] ≥ i/2 then
 7        i ← i − border_x[i] + i mod (i − border_x[i])
 8     Push(S, i)
 9     i ← border_x[i]
10  b ← Pop(S)
11  while |S| > 0 do
12     listPos ← pos(x[0 .. b − 1], x[0 .. Top(S) − 1]) ∪ (Top(S))
13     while |S| > 1 and isCovering(listPos, b) do
14        Pop(S)
15        listPos ← pos(x[0 .. b − 1], x[0 .. Top(S) − 1]) ∪ (Top(S))
16     if isCovering(listPos, b) then
17        return x[0 .. b − 1]
18     else b ← Pop(S)
19  return x[0 .. b − 1]
```

Figure 3. Classical computation of the shortest cover of string x of length m.

For a string to be a cover, every occurrence of the border must occur in a way that ensures complete coverage of x. Here, the smallest border, `a`, is not sufficient to cover `aba`. However, the border `aba` does cover the full string x. Thus, the shortest cover of x is `aba`. This result confirms that the minimal repeating unit that effectively covers x is of length 3.

The classical algorithm for computing the shortest cover of a string is given in Figure 3. It first computes the border array and then processes the sequence of borders using a stack S (Lines 5-10). When a border w of a prefix $x[0\,..\,n-1]$ has length $i > n/2$, its length is adjusted as: $n - i + n \bmod (n - i)$ (Lines 6-7). This effectively reduces w to uu', where $w = u^r u'$ and u' is a prefix of u.

The algorithm employs function IsCovering$(listPos, b)$ at lines 13 and 16 to verify whether a border b covers the list of positions *listPos*. Function IsCovering takes a list L of integers and an integer b and checks that the difference between consecutive elements in L is less than b. Function *pos* (lines 12 and 15) can be implemented using any linear-time string matching algorithm.

3 Computing the period of a string from its `CDS` representation

We now present an efficient method for computing the period of a string using its `CDS` representation. We first consider the case where the alphabet consists of only two distinct characters, meaning $\sigma = |\Sigma| = 2$, thus $\Sigma = \{a, b\}$. Let a be the only pivot character. Let x be a string of length m. The function δ denotes the positions of occurrences of a designated pivot character in x. The `CDS` representation of x is denoted by $\bar{x}$, and its length is given by $\bar{m} = |\bar{x}|$.

The key idea is that the periodic structure of $\bar{x}$ provides enough information to determine the period of x. Before proving this, we present two auxiliary results that establish the fundamental relationships between x and its CDS representation.

From any position $\delta(i)$ of the pivot character a, all positions in δ greater than $\delta(i)$ correspond to $\delta(i)$ plus a sum of elements from $\bar{x}$. These positions also contain a.

Lemma 1. *Let $x[\delta(i)] = a$ for $0 \leq i \leq \bar{m} - 1$. Then:*

$$x[\delta(i) + \sum_{j=i}^{k} \bar{x}[j]] = a, \quad \text{for } i \leq k \leq \bar{m} - 1.$$

Proof. Expanding the sum of distances between consecutive pivot occurrences, we obtain:

$$x[\delta(i) + \sum_{j=i}^{k} \bar{x}[j]] = x[\delta(i) + (\delta(i+1) - \delta(i)) + \cdots + (\delta(k+1) - \delta(k))].$$

Since $\delta(k+1)$ corresponds to the position of the pivot character, it follows that: $x[\delta(k+1)] = a$.

A similar result holds for positions preceding a pivot occurrence.

Lemma 2. *Let $x[\delta(i)] = a$ for $0 \leq i \leq \bar{m} - 1$. Then:*

$$x[\delta(i) - \sum_{j=k}^{i-1} \bar{x}[j]] = a, \quad \text{for } 0 \leq k \leq i - 1.$$

Proof. By summing the distances between previous occurrences of the pivot character, we obtain:

$$x[\delta(i) - ((\delta(k+1) - \delta(k)) + \cdots + (\delta(i) - \delta(i+1)))] = x[\delta(k)].$$

Since $\delta(k)$ corresponds to an occurrence of the pivot character, we conclude that: $x[\delta(k)] = a$.

Observation 1 *Let $k = \sum_{i=0}^{per(\bar{x})-1} \bar{x}[i]$. By definition of $per(\bar{x})$, it follows that:*

$$\sum_{i=0}^{per(\bar{x})-1} \bar{x}[j+i] = k, \quad \text{for } 1 \leq j \leq \bar{m} - per(\bar{x}).$$

Let $x[0]$ be the pivot character. If the first and last character of x are the same ($x[0] = x[m-1]$), the period of x is determined directly from the sum of the first $per(\bar{x})$ elements of $\bar{x}$. If they are different ($x[0] \neq x[m-1]$), additional conditions must be checked.

Figure 4 illustrates the computation of the period for $x =$ `abaababaaba`. We obtain the period of x by summing the first three elements of $\bar{x}$, yielding $per(x) = 2{+}1{+}2 = 5$. This example demonstrates how the periodic properties of $\bar{x}$ provide a direct way to compute the period of x efficiently.

If $x[0] \neq x[m-1]$, the structure of the borders must be analyzed further. Let $x = auav$, where $u \in \Sigma^*$ and $v = b^k$ does not contain any occurrences of a. The border of x can be computed by examining the borders of aua.

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	a	a	b	a	b	a	a	b	a
$\bar{x}[i]$	2	1	2	2	1	2					
$border_{\bar{x}}[i]$	-1	0	0	1	1	2	3				

Figure 4. Border array of the CDS representation of $x =$ abaababaaba of length 11 with pivot a. Then $per(\bar{x}) = 6 - border_{\bar{x}}[6] = 3$, thus $per(x) = \bar{x}[0] + \bar{x}[1] + \bar{x}[2] = 2 + 1 + 2 = 5$.

Lemma 3. *The border of auav is given by $w'v$, where w' is the longest border of aua followed by $v' = b^{k'}$ such that $k \leq k'$, or the border of auav is empty if there is no such border w' of aua.*

Proof. Assume that w is the border of $auav$. Since v does not contain any a, w should be equal to $w'v$. Since $w'v$ is a prefix and a suffix of $uauv$, w' is a prefix and a suffix of aua, and thus a border of aua. Also it implies that aua has a prefix $w'v^{k'}$ where $k \leq k'$. Assume by contradiction that there exists a border $w"$ of aua such that $w"v^{k"}$ is a prefix of aua with $k \leq k"$ and $|w"| > |w'|$. Then $w"v$ would be a border of $auav$ longer than its border, a contradiction.

Figure 5 illustrates the CDS representation of the string $x =$ abbababbabb and the steps involved in computing its period.

When x is of the form $auab^k$, checking if a border of aua is followed by $b^{k'}$ with $k \leq k'$ can be determined in constant time.

i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	a	b	b	a	b	a	b	b	a	b	b
$\bar{x}[i]$	3	2	3								
$border_{\bar{x}}[i]$	-1	0	0	1							

Figure 5. Border array of the CDS representation of $x =$ abbababbabb $=$ abbababbab^2 of length 11 with pivot a thus $k = 2$. $border_{\bar{x}}[3] = 1$ but $\bar{x}[border_{\bar{x}}[3] = 1] = 2 \leq k = 2$. $border_{\bar{x}}[1] = 0$ and $\bar{x}[border_{\bar{x}}[1] = 0] = 3 > k = 2$. Since we use the border of $\bar{x}$ of length $border_{\bar{x}}[1] = 0$, we use the period of $\bar{x}$ which is equal to $|\bar{x}| - border_{\bar{x}}[1] = 3$, then we sum the first 3 elements of $\bar{x}$ to get $per(x) = \bar{x}[0] + \bar{x}[1] + \bar{x}[2] = 3 + 2 + 3 = 8$.

Let us now consider the case where the alphabet contains more than two distinct characters. Suppose that x is of the form $x = auav$, where $u \in \Sigma^*$ and v is a substring that contains no occurrences of the pivot character a. In this scenario, the longest candidate border of x computed by the previous algorithm must be explicitly verified to determine whether it is a valid border of x. This candidate border should be followed by a substring v', which, like v, contains no occurrences of a and is at least as long as v, i.e., $|v'| \geq |v|$. The algorithm then checks whether the candidate border occurs both as a prefix and a suffix of x. If this condition holds, the corresponding border length enables to compute a valid period for x. If not, the process continues iteratively, examining the next longest candidate border and its associated suffix v', until either a valid period is found or no further non-empty borders exist. If no valid period is identified through this process, it follows that x is borderless, and its period is simply m, the length of the entire string.

The algorithms presented in Figure 6 efficiently implement this method. Since several prefix sums of $\bar{x}$ may be required in the computation, the algorithm uses a

```
PerCDS(x, m)
 1  x̄ ← CDS version of x
 2  m̄ ← |x̄|
 3  k ← length of the longest
        suffix of x that
        does not contain any a
 4  bor ← border_x̄
 5  s[0] ← x̄[0]
 6  for i ← 1 to m̄ − 1 do
 7     s[i] ← s[i − 1] + x̄[i]
 8  (b, b̄) ←
      BorCDS(x, m, x̄, m̄, k, s, bor)
 9  period ← m − b
10  return period
```

```
BorCDS(x, m, x̄, m̄, k, s, bor)
 1  i ← m̄
 2  while i > −1 do
 3     i ← bor[i]
 4     while i > −1 and x̄[i] ≤ k do
 5        i ← bor[i]
 6     if i > −1 then
 7        p̄ ← m̄ − i
 8        p ← s[p̄ − 1]
 9        b ← m − p
10        if x[0 .. b − 1] =
             x[m − b .. m − 1] then
11           return (b, i)
12  return (0, −1)
```

Figure 6. Computation of the period of a string x of length m with its CDS representation $\bar{x}$ of length $\bar{m}$ (left) computed using the border array of $\bar{x}$ and its cumulative sum array s (right). The variable k corresponds to the number of non-pivot symbols after the rightmost pivot symbol in x. The algorithm BorCDS returns the length of the border of x and the length of the border of $\bar{x}$ that corresponds to the period of $\bar{x}$ that can enable to compute the period of x (this is needed when computing covers).

precomputed cumulative sum array s over $\bar{x}$. This preprocessing step ensures that the necessary sums can be retrieved in constant time, optimizing the overall efficiency of the method. As in the case of exact string matching with sampling, the method only computes border candidates that need to be checked (line 10 of Algorithm BorCDS).

Theorem 4. *Algorithm* PerCDS(x, m) *computes the period of a string x of length m in time $O(\bar{m})$ and in extra space $O(m)$.*

Proof. The correctness of the algorithm comes from Lemmas 1 to 3. The time complexity comes from the fact that the inner loop of algorithm BorCDS can only be executed $O(\bar{m})$-time and that the comparison in line 10 can be performed in constant time using Range Minimum Queries on a Longest Common Prefix array on the suffixes of x. The extra space complexity comes from some constants and the arrays $border_{\bar{x}}$ and s that require $O(\bar{m})$ space and the $O(m)$ space needed to compute the RMQs in constant time.

In practice, the extra space can be reduced to $O(\bar{m})$ by performing the comparisons in a brute-force way. The time complexity then becomes $O(m\bar{m})$. Experimental results show that this method is faster than the classical one.

4 Computing the shortest cover of a string from its CDS representation

The classical computation of the shortest cover of a string x relies on its border array. Since borders play a fundamental role in determining covers, the approach used to compute borders in the CDS representation can be adapted to efficiently determine the shortest cover as well. A slight modification of Algorithm borCDS in Figure 6 allows to compute the lengths of all the non-empty non-periodic borders in decreasing order of their length.

```
BordersCDS(x, m, x̄, m̄)
 1  border_x̄ ← Borders(x̄, m̄)
 2  B ← empty list
 3  k ← length of the longest suffix of x that does not contain any a
 4  s[0] ← x̄[0]
 5  for i ← 1 to m̄ − 1 do
 6     s[i] ← s[i − 1] + x̄[i]
 7  i ← m̄
 8  (b, i) ← BorCDS(x, m, x̄, m̄, k, s, border_x̄)
 9  if b = 0 then
10     return B
11  while i > −1 and s[i] ≥ b do
12     i ← i − 1
13  m̄ ← i + 1
14  if i < 0 then
15     return (b)
16  while i > −1 do
17     (b′, i′) ← BorCDS(x, m, x̄, m̄, k, s, border_x̄)
18     if b′ < b/2 then
19        B ← B ⊕ (b)
20     b ← b′
21     while i > −1 and s[i] ≥ b do
22        i ← i − 1
23     m̄ ← i + 1
24  if b′ ≠ 0 then
25     B ← B ⊕ (b′)
26  return B
```

Figure 7. Computation of the non-empty non-periodic borders of string x of length m using its CDS representation. Lines 4-6 compute the cumulative sums. Lines 8-25 consider the borders of x in decreasing order of their length using the algorithm BorCDS described in Fig. 6.

Since every cover of a string must also be a border, the computation of all non-empty non-periodic borders represents the fundamental step for determining the shortest cover. Figure 7 introduces the procedure used to extract these borders from the CDS representation of the input string. The algorithm returns the candidate border lengths in decreasing order, allowing them to be processed efficiently without explicitly reconstructing the full string.

Once the set of candidate borders has been obtained, the shortest cover can be determined by verifying which border satisfies the covering property. Figure 8 describes this second phase. The algorithm processes the candidate borders produced by bordersCDS, computes the occurrences of each candidate prefix, and checks whether every position of the string is covered by at least one occurrence. The first candidate satisfying this condition corresponds to the shortest cover of the string.

5 Experimental results

We now present experimental results to evaluate the performance of the algorithms proposed in this paper. Our objective is to assess the computational efficiency of the methods that leverage the CDS representation for computing the period and shortest

```
ShortestCoverCDS(x, m, x̄, m̄)
 1  borders ← BordersCDS(x, m, x̄, m̄)
 2  S ← EmptyStack()
 3  Push(S, m)
 4  for bor ∈ borders do
 5     Push(S, bor)
 6  b ← Pop(S)
 7  while |S| > 0 do
 8     listPos ← pos(x[0..b − 1], x[0..Top(S) − 1]) ∪ (Top(S))
 9     while |S| > 1 and IsCovering(listPos, b) do
10        b ← Pop(S)
11        listPos ← pos(x[0..b − 1], x[0..Top(S) − 1]) ∪ (Top(S))
12     if IsCovering(listPos, b) then
13        return x[0..b − 1]
14     else b ← Pop(S)
15  return x[0..b − 1]
```

Figure 8. Computation of the shortest cover of a string x of length m using its `CDS` representation. The algorithm exploits the set of candidate borders computed by BordersCDS and verifies the covering property through occurrences of each candidate prefix. Lines 6-14 consider the non-empty non-periodic borders of x in increasing order of their length.

cover of a string. Note that we did not use suffix arrays and RMQs to compute string comparisons in constant time but that such comparisons were naively performed.

The algorithms have been implemented in the `C` programming language and tested using the Smart tool [4]. The experiments were conducted on a MacBook Pro equipped with a 2.7 GHz Intel Core i7 processor, four cores, 16 GB of 2133 MHz LPDDR3 RAM, a 256 KB L2 cache, and an 8 MB L3 cache. The compilation was performed using the `gcc` compiler with the `-O3` optimization flag to ensure optimal execution speed. [1]

To obtain statistically significant results, each algorithm was executed over 10,000 different runs, and the average execution time was recorded. The dataset used in our experiments consists of 100MB of genomic text extracted from the *Pizza and Chili* corpus [12], which is widely adopted in the field of string processing research. The experiments were performed on input strings of varying lengths, ranging from 10^4 to 10^6 characters.

For correctness, the pivot character selected for constructing the `CDS` representation was always the first character of the input string. The additional memory required to store the `CDS` representation varied between 3% and 15% of the size of the original string, depending on the structure of the input data.

The results in Figure 9 indicate that the method for computing the period using the `CDS` representation achieves a speedup between 81% and 85% compared to the standard approach.

Similarly, Figure 9 also presents the execution times for computing the shortest cover of a string using both the standard method and the optimized `CDS`-based approach. The results show that the `CDS` representation leads to a significant performance gain, achieving speedups between 34% and 40%.

[1] The Smart tool is available for download at `https://github.com/ostafen/smart/tree/develop`.

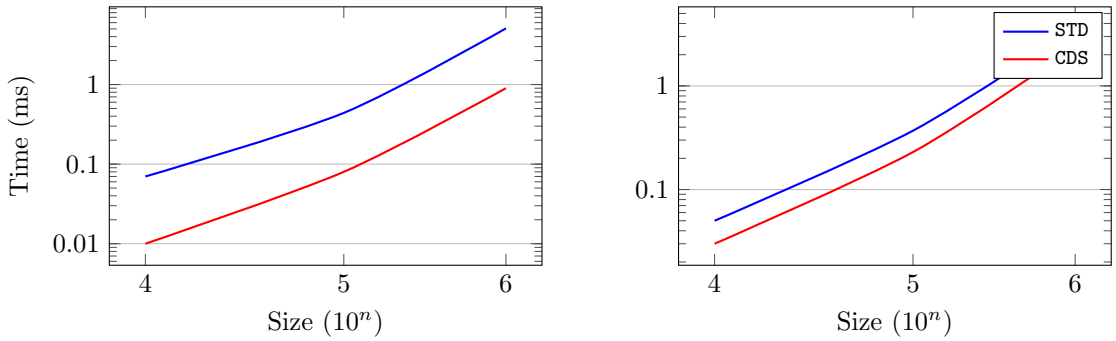

Figure 9. Execution times over 10,000 runs: (Left) shortest cover, (Right) period.

6 Conclusion

In this paper, we demonstrated that the partial index built using the *Characters-Distance-Sampling* (CDS) algorithm, originally designed to enhance the efficiency of string matching algorithms, can also be effectively utilized to compute regularities in strings. This novel approach leverages the compact and structured nature of the CDS representation to extract fundamental properties of strings in a computationally efficient manner.

Specifically, we introduced optimized methods for solving two well-studied problems in the field of string regularities: the computation of the *period* and the detection of *quasiperiods* or *covers*. By adapting classical techniques to work with the CDS representation, we developed algorithms that significantly outperform traditional approaches. Experimental results confirm the effectiveness of our methods, showing speedups ranging from 38% to 43% for period computation and from 81% to 85% for shortest cover detection. These improvements demonstrate the potential of the CDS representation not only for accelerating string matching but also for efficiently identifying structural properties of strings.

Our findings open the door for further research into the use of CDS-based methods for detecting other types of regularities in strings. Future work could explore the application of this approach to problems such as the computation of *seeds*, *k-covers*, or other forms of repetitive structures. Investigating the theoretical bounds of this technique, as well as its practical implications in bioinformatics, data compression, and text processing, could lead to new advancements in the field.

The same techniques can be easily adapted to strings given in other kinds of representations such as the run-length encoding.

By demonstrating that a representation initially designed for string matching can be repurposed to analyze structural properties of strings, this work provides a new perspective on the versatility of distance-based indexing techniques. The promising results obtained in this study suggest that CDS-based methods could be further developed to enhance the efficiency of a wide range of string-processing applications.

References

1. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* Theor. Comput. Sci., 119(2) 1993, pp. 247–265.
2. A. Apostolico, M. Farach, and C. S. Iliopoulos: *Optimal superprimitivity testing for strings.* Inf. Process. Lett., 39(1) 1991, pp. 17–20.

3. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on strings*, Cambridge University Press, 2007.
4. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Prague Stringology Conference, 2016, pp. 99–111.
5. S. Faro, T. Lecroq, and F. P. Marino: *Optimal character distance sampling for exact string matching through set cover reformulation.* IEEE Access, 14 2026, pp. 13910–13920.
6. S. Faro and F. P. Marino: *Reducing time and space in indexed string matching by characters distance text sampling*, in Prague Stringology Conference, 2020, pp. 148–159.
7. S. Faro, F. P. Marino, and A. Moschetto: *Beyond Horspool: A comparative analysis in sampled matching*, in Prague Stringology Conference, 2024, pp. 16–26.
8. S. Faro, F. P. Marino, A. Moschetto, A. Pavone, and A. Scardace: *The great textual hoax: Boosting sampled string matching with fake samples*, in 12th International Conference on Fun with Algorithms, FUN 2024, June 4-8, 2024, Island of La Maddalena, Sardinia, Italy, A. Z. Broder and T. Tamir, eds., vol. 291 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 13:1–13:17.
9. S. Faro, F. P. Marino, and A. Pavone: *Efficient online string matching based on characters distance text sampling.* Algorithmica, 82(11) 2020, pp. 3390–3412.
10. S. Faro, F. P. Marino, and A. Pavone: *Enhancing characters distance text sampling by condensed alphabets*, in Italian Conference on Theoretical Computer Science, 2021, pp. 1–15.
11. S. Faro, F. P. Marino, and A. Pavone: *Improved characters distance sampling for online and offline text searching.* Theor. Comput. Sci., 946 2023, p. 113684.
12. P. Ferragina and G. Navarro: *Pizza&Chili, Available at:* `pizzachili.dcc.uchile.cl`, 2005.
13. N. Mhaskar and W. F. Smyth: *String covering: A survey.* Fundam. Inform., 190(1) 2022, pp. 17–45.
14. W. F. Smyth and M. Yusufu: *Computing regularities in strings*, in IEEE International Conference on Computer Science and Information Technology, 01 2009, pp. 298–302.

Updating Permutation-Compressed Hypergraphs

Enno Adler, Stefan Böttcher, and Rita Hartel

Paderborn University,
Warburger Straße 100, Paderborn, Germany
{enno.adler, stefan.boettcher, rita.hartel}@uni-paderborn.de

Abstract. Hypergraphs model complex, non-binary relationships such as co-authorships, social group memberships, and recommendation systems. Like traditional graphs, hypergraphs can grow large, posing challenges for storage, transmission, and query performance. Moreover, the data can change as new papers are published, new social groups form, existing social groups accept or remove members, and new products are published. In this paper, we extend the hypergraph compression method HyperCSA to perform updates of the compressed hypergraph. Dynamic HyperCSA computes the exact compressed hypergraph representation that the steps *decompress-update-compress* of HyperCSA produce, but is orders of magnitude faster. Thereby, dynamic HyperCSA achieves the same compression ratios of 26% to 79% as HyperCSA on the modified, real-world hypergraphs and the query performance of being 6 to 40 times faster on real-world hypergraphs than other approaches.

Keywords: hypergraph compression, update operations, permutation-based compression

1 Introduction

Hypergraphs are powerful tools for modeling complex, non-binary relationships in data. Unlike traditional graphs, where edges connect only two nodes, hypergraphs allow edges to connect any number of nodes, enabling a more accurate representation of real-world systems. This modeling flexibility has made hypergraphs useful in a range of domains, including social networks [12], music [11] and news [6] recommendation systems, academic authorship networks, and tagged data repositories from online platforms [4]. To store and query large datasets, Adler et al. [1] recently introduced HyperCSA, a permutation-based hypergraph compression scheme based on the compressed suffix array by Sadakane [9]. However, HyperCSA cannot modify the underlying hypergraph yet. In this paper, we show how to alter the compressed representation of HyperCSA to reflect changes in the hypergraph data.

Herein, our main contributions are as follows:

- Algorithms[1] to add and remove nodes or edges from a hypergraph and algorithms to add and remove a single node from an edge in a hypergraph compressed by HyperCSA.
- Theoretical worst-case runtimes for all mentioned algorithms.
- A practical evaluation showing that the proposed algorithms significantly outperform the naive decompress-update-compress approach of updating a hpergraph compressed with HyperCSA.

[1] An implementation of all algorithms proposed in this paper is available at `https://github.com/adlerenno/hypercsa`.

Enno Adler, Stefan Böttcher, Rita Hartel: Updating Permutation-Compressed Hypergraphs, pp. 97–111.
Proceedings of PSC 2026, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-07576-0 © Czech Technical University in Prague, Czech Republic

2 Preliminaries

A *hypergraph* is a pair $G = (V_G, E_G)$ with nodes $V_G = \{0, 1, \ldots, p\} \subset \mathbb{N}_0$ and edges $E_G = \{e_0, \ldots, e_{m-1}\}$. For every hyperedge $e_i = \{v_0, \ldots, v_{n-1}\} \subseteq V_G$, $e_i \neq \emptyset$ and $\text{rank}(e_i) = |e_i| = n$. We call all hyperedges in E_G edges regardless of their rank. HyperCSA can also handle hypergraphs that contain the same edge multiple times. In Figure 1, we visualize our ongoing example.

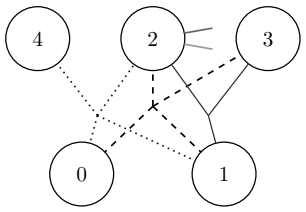

Figure 1: The example hypergraph H with nodes $V_H = \{0, 1, 2, 3, 4\}$ and hyperedges $E_H = \{\{0, 1, 2, 3\},\ \{1, 2, 3\},\ \{2\},\ \{0, 1, 2, 4\},\ \{2\}\}$.

We define $\text{degree}(v) = |\{e \in E_G : v \in e\}|$, $R_G = \max\{\text{rank}(e) : e \in E_G\}$, $\Delta_G = \max\{\text{degree}(v) : v \in V_G\}$, and $M_G = \sum_{e \in E_G} \text{rank}(e) = \sum_{v \in V} \text{degree}(v)$. For the example hypergraph H, we have $\text{degree}(2) = 5$, $R_H = 4$, $\Delta_H = 5$, and $M_H = 13$.

A string T of length $|T|$ over an alphabet Σ with total order $<$ is a sequence of characters $T[i] \in \Sigma$ for $0 \leq i < |T|$, so $T = T[0] \cdot T[1] \cdots T[|T| - 1]$. We write ϵ for the empty string, $T[i, j]$ for the substring $T[i] \cdot T[i + 1] \cdots T[j]$ of T, $T[i, j] = \epsilon$ if $i > j$, and $T[i..] = T[i, |T| - 1]$ for the suffix starting at position i. We call a position i an S-type, if $T[i] < T[i + 1]$, and L-type otherwise, also if $i = |T| - 1$.

We define $\text{rank}_T(x, c) = |\{i < x : T[i] = c\}|$ as the number of times c occurs in the first x characters of T (excluding the current position). Next, let $\text{select}_T(i, c) = \min\{x < |T| : \text{rank}_T(x, c) = i\} - 1$ be the position of the i-th c in T.

The suffix array SA [8] of a string T is a permutation of $\{0, \ldots, |T| - 1\}$ such that the i-th smallest suffix of T is $T[SA[i]..]$ with $i \in \{0, \ldots, |T| - 1\}$. As permutations are invertible, we denote the inverse of the suffix array by SA^{-1}. We use the term *T-position* for a position in T and *SA-position* for a position in SA. The compressed suffix array CSA by Sadakane [9] of a string T consists of a bitvector D of length $|T|$ and a permutation Ψ_{CSA} of $\{0, \ldots, |T| - 1\}$ (and other here omitted data structures). The bitvector D is a succinct representation for T using the suffix array SA: $D[i] = 1$ marks the first suffix in the sorted suffixes of T that start with a new character, so $D[i] = 1$ if and only if $T[SA[i]] > T[SA[i - 1]]$ or $i = 0$. We assume that all symbols of Σ have at least one occurrence in T. By this assumption, we get $T[SA[i]] = \text{rank}_D(i, 1) - 1$. $\Psi_{CSA}[i]$ stores the SA-position j that points to the next T-position $SA[i] + 1$ after the T-position $SA[i]$ at SA-position i. In one term, $\Psi_{CSA}[i] = SA^{-1}[(SA[i] + 1) \mod |T|]$. Ψ_{CSA} is a cyclic permutation and by computing the ranks on D for the SA-positions i in order of the cycle of Ψ_{CSA}, we retrieve a rotation of T.

The HyperCSA approach by Adler et al. [1] is inspired by RDFCSA by Brisaboa et al. [2], which is a self-index for the triples representing an RDF graph. As HyperCSA is a generalization of RDFCSA, all presented algorithms can be adapted for RDFCSA. HyperCSA's construction consists of three steps: First, we sort the nodes in an edge in ascending order and then, we sort the edges in lexicographically descending order. Thereafter, we concatenate all edges into one string T. For our example, we get

T-position i	0	1	2	3	4	5	6	7	8	9	10	11	12	
$T[i]$	2	2	1	2	3	0	1	2	4	0	1	2	3	
$SA[i]$	9	5	10	2	6	1	0	11	3	7	12	4	8	
SA-position i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$D[i]$	1	0	1	0	0	1	0	0	0	0	1	0	1	1
$\text{node}_D(i)$	0	0	1	1	1	2	2	2	2	2	3	3	4	5
$\Psi_{CSA}[i]$	2	4	7	8	9	3	5	10	11	12	1	0	6	
$\Psi[i]$	2	4	7	8	9	5	6	10	11	12	0	3	1	

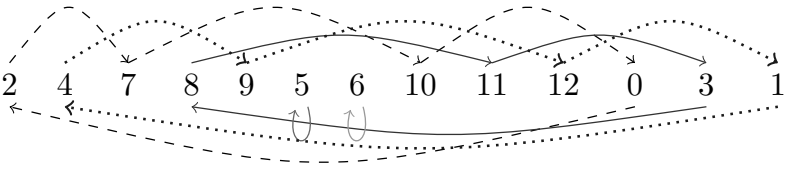

Figure 2: HyperCSA data structure for the hypergraph H. The node at index i is $\text{node}_D(i) = \text{rank}_D(i+1, 1) - 1$. The black numbers in Ψ_{CSA} are the numbers adjusted by the third step of HyperCSA's construction. The style of every edge in Ψ is the same as in Figure 1. The arrows of SA-positions with $\Psi[i] \leq i$ are drawn below the numbers, all others are drawn above.

$T = 2|2|123|0124|0123$, where the vertical bars are inserted as visual aid to identify the edges. Second, we compute the compressed suffix array of T and append a 1 to D for easier querying. We show D and Ψ_{CSA} of the compressed suffix array for our example T in Figure 2. Third, we modify Ψ_{CSA} such that each edge gets its own cycle in the permutation Ψ. We depict these cycles in Figure 2 and provide the algorithm in Agorithm 1. The adjustment from Ψ_{CSA} to Ψ only requires one change $\Psi_{CSA}[j] = i$ for every hyperedge e, where j is the single L-type position of e and i is the smallest position that belongs to e in Ψ_{CSA}.

Algorithm 1: HyperCSA.*adjust*Ψ

```
   Input: Ψ_CSA (in the following renamed to Ψ)
 1 current_position ← Ψ[0]
 2 next ← Ψ[current_position]
 3 last_first_node_position ← 0
 4 while current_position is not 0 do
 5 |   if next < current_position then
 6 |   |   Ψ[current_position] ← last_first_node_position
 7 |   |   last_first_node_position ← next
 8 |   current_position ← next
 9 |   next ← Ψ[next]
```

We define $\text{node}(i) = \text{rank}_D(i+1, 1) - 1$ and $\text{edge}(i) = \{\text{node}_D(\Psi^k[i]) : k = 0, 1, \dots\}$. We call the SA-position i a representative of the edge $\text{edge}(i)$ for all $i < |T|$. For example, 0, 2, 7, and 10 are all representatives of the edge $\{0, 1, 2, 3\}$ in Figure 2.

For $v \in V$, let $S(v) = \{\text{select}_D(v+1,1), \ldots, \text{select}_D(v+2,1)-1\}$ be the set of the SA-positions i where $T[SA[i]] = v$. For example, for $v = 1$, we have $S(1) = \{2,3,4\}$. By construction of the (compressed) suffix array, every $S(v)$ is a contiguous set of natural numbers, as can be seen in Figure 2.

HyperCSA uses the δ-code by Elias [3] on the differences $\Psi[i] - \Psi[i-1]$ between two values to encode Ψ. This works well because for node $v \in V$, $\Psi[S(v)]$ is a contiguous increasing integer sequence [1]. However, for this paper, we will work with the numerical Ψ without any encoding.

3 State of the Art on Updateable Hypergraph Compression

Many hypergraph compression approaches do not have an update algorithm for their compressed structure. This includes the approaches HybHuff of Zhao et al. [13] and the Hygra implementation using Ligra by Shun [10]. Liu et al. [7] propose an approach to reorder the elements of an incidence list and thereafter compress it. In comparison to the other mentioned methods, Liu et al. provides a maintenance algorithm that can be used to update the order of elements in the incidence lists after modification. The worst-case runtime of their compression algorithm and their maintenance algorithm is

$$\mathcal{O}\left(\sum_{u \in V} (mdeg(u))^2 + n^2\right)$$

with $mdeg(u)$ being the sum of the ranks of hyperedges that are incident to the node $u \in V$. Regardless of the type of modification, their approach always uses the same maintenance algorithm. In Table 1, we show all worst-case runtime bounds for the operations of HyperCSA and compare them to the worst-case runtime bounds for the incidence list. We highlight that every update operation of HyperCSA has a lower worst-case bound than the approach of Liu et al. [7].

4 Dynamic HyperCSA

We present the modification operations in the following order:

1. deleting an edge,
2. removing a node from an edge,
3. inserting a node to an edge,
4. adding an edge,
5. deleting a node, and
6. adding a node.

We remark that modifying the HyperCSA structure normally renders all prior by queries obtained representatives useless.

We will use our example from Figure 1 of a hypergraph H containing the hyperedges $\{2\}$, $\{2\}$, $\{0,1,2,3\}$, $\{1,2,3\}$, and $\{0,1,2,4\}$. We have $D = 10100100001011$ and $\Psi = [2,4,7,8,9,5,6,10,11,12,0,3,1]$.

4.1 Deleting an Edge

In a list of hyperedges, deleting the third edge is easy and unambiguous. However, in the HyperCSA data structure, it is not clear, which edge is the third. Therefore, any representative p of an edge edge(p) can be used to call deleteEdge(p).

method	incidence list	HyperCSA
construct	$\mathcal{O}(M_G)$	$\mathcal{O}(\lvert E_G\rvert R_G(\lg(R_G)+\lg(\lvert E_G\rvert)))$
edge(p)	$\mathcal{O}(\text{rank}(\text{edge}(p)))$	$\mathcal{O}(\text{rank}(\text{edge}(p)))$
decompress(G)	$\mathcal{O}(M_G)$	$\mathcal{O}(M_G)$
degree(v)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
rank(i)	$\mathcal{O}(1)$	$\mathcal{O}(\text{rank}(i))$
contains(v)	$\mathcal{O}(\text{degree}(v))$	$\mathcal{O}(\text{degree}(v))$
contains(Q)	$\mathcal{O}(n + R_G \min_i(\text{degree}(v_i)))$	$\mathcal{O}(n + R_G \min_i(\text{degree}(v_i)))$
exists(Q)	$\mathcal{O}(n \min_i(\text{degree}(v_i)))$	$\mathcal{O}\left(\sum_{i=0}^{n} \log(\text{degree}(v_i))\right)$
deleteEdge(i)	$\mathcal{O}(R_G \cdot \Delta_G)$	$\mathcal{O}(M_G)$
removeNode(n, i)	$\mathcal{O}(\text{rank}(i) + \text{degree}(n))$	$\mathcal{O}(M_G + R_G^2 \cdot \Delta_G)$
insertNode(n, i)	$\mathcal{O}(\text{rank}(i) + \text{degree}(n))$	$\mathcal{O}(M_G + R_G^2 \cdot \Delta_G)$
addEdge(e)	$\mathcal{O}(\text{rank}(e))$	$\mathcal{O}(M_G)$
deleteNode(n)	$\mathcal{O}(M_G)$	$\mathcal{O}(\text{degree}(n) \cdot (M_G + R_G^2 \cdot \Delta_G))$
addNode(n)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 1: Comparison of worst-case runtimes of the incidence list and HyperCSA. Runtimes of queries are included from Adler et al. [1]. The space requirements are $2M_G + |E_G| + |V_G|$ integers for the incidence list and $M_G(H_0(T) + 1) + \mathcal{O}(M_G \log(H_0(T)))$ bits for HyperCSA (with $H_0(T)$ being the zero-order empirical entropy of T). We use Q as a shortcut for $\{v_1, \ldots, v_n\}$.

We delete an edge containing position p in two steps: First, we mark all positions we are going to delete. Second, we shorten the HyperCSA data structure. Both steps can be done for multiple deletions at once, but we present the algorithm for only one deletion.

First, we create a bitvector Z with $Z[i] = 1$ if $i = \Psi^k[p]$ for some k and $Z[0] = 0$ otherwise. Thereby, Z encodes the positions of the edge edge(p), which are the positions we delete from HyperCSA.

As an example, we show deleteEdge(3) on the hypergraph H of Figure 2. Because $\Psi^1[3] = 8$, $\Psi^2[3] = \Psi[8] = 11$, and $\Psi^3[3] = \Psi[11] = 3$, we get $Z = 0001000010010$. In the following paragraphs, we use D', Ψ', and Ψ'' for presenting and unambiguously referencing steps of our approach. However, our implementation does not generate new arrays for them. Our implementation alters the existing D and Ψ array.

The second step is to compute a new D' from D: The length of D' is the length of D minus rank(e), the rank of the edge we want to delete, which is the length of the cycle containing p. Retrieving D' is easy: We always append the next value to D', so we build D' left to right starting at position 0. We iterate over $Z[i]$ and $D[i]$: If $Z[i] = 0$, we append $D[i]$ to D'. If $Z[i] = 1$, we skip the value of $D[i]$ except if $D[i] = 1$, because $D[i] = 1$ indicates the start of the interval of the next node v. In this case, we add a 1 and skip the following position if it fulfills $D[i + 1] = 0$. If $D[i+1] = 1$, our call deleteEdge(p) requires reducing the degree of the node v to zero. Deleting the 1 of its span would cause all larger node IDs to shrink by 1, which would alter the data significantly. There is no way to compensate for this in HyperCSA, because this deletion violates the minimality of V. Therefore, we abort the deletion with an exception in this case. We discuss a solution in Section 4.6.

In our example, we get $D' = 10101000111$. In words, the nodes 0 and 1 occur two times, the node 2 is connected to four hyperedges, and the node 3 and 4 to only one hyperedge each. The information about which node is connected to which hyperedge is saved using Ψ.

The third step is to obtain a new Ψ'. This works similar to computing D' from D, but is even more straightforward: We iterate over $Z[i]$ and $\Psi[i]$ at the same time and we append $\Psi[i] - \text{rank}_Z(\Psi[i], 1)$ to Ψ' if and only if $Z[i] = 0$. Adjusting $\Psi[i]$ by subtracting $\text{rank}_Z(\Psi[i], 1)$ is necessary because we deleted $\text{rank}_Z(\Psi[i], 1)$ positions before position $\Psi[i]$ and we need to compensate this in Ψ'.

For example in Step $i = 2$, we have $\Psi[2] = 7$.

$$\begin{aligned}\Psi'[2] &= \Psi[2] - \text{rank}_Z(\Psi[2], 1)\\ &= 7 - \text{rank}_Z(7, 1)\\ &= 7 - 1 = 6.\end{aligned}$$

Overall, we obtain $\Psi' = [2, 3, 6, 7, 4, 5, 8, 9, 0, 1]$.

Clearly, all data structures use $\mathcal{O}(M_G)$ space and we only iterate once over the deleted edge, D, and Ψ, so deleting the edge takes $\mathcal{O}(M_G)$ time.

4.2 Removing a Node from an Edge

The call removeNode(p, v) supposes us to delete the node v from the edge $e = \text{edge}(p)$. As an example, we will show removeNode$(3, 2)$. First, we check if e contains v. This is achieved by iterating over the positions of the edge starting at p by applying Ψ recursively on the result. For such a position i, we determine the corresponding node node(i) and check if node$(i) = v$. If we reach p again and we have not found the node v, there is no node to delete and we can finish. If we found v, we also acquired position q representing the connection of edge e to node v. In our example, we retrieve the edge at positions $(3, 8, 11)$, and $q = 8$.

Next, we calculate the new D' from D. Mostly, we copy the values from D, but we skip the value at position q if $D[q] = 0$. If $D[q] = 1$ and $D[q + 1] = 0$, we put a 1 at $D'[q]$ and skip the position $q + 1$ in D. If $D[q] = D[q + 1] = 1$, deleting q reduces the degree of v to zero. Like for deleting edges, this is not allowed and we abort. We discuss a solution in Section 4.6.

In the example, removeNode$(3, 2)$ means we remove node 2 from the edge $(1, 2, 3)$, which results in the new edge $(1, 3)$. We adjust D to $D' = 1010010001011$, where we deleted one 0 after the third 1, which means that the node 2 occurs one fewer time.

The third step is adjusting Ψ. The idea is simple: The jump pointing to q should now point to $\Psi[q]$, so the cycle skips position q afterwards. The main work on adjusting Ψ results from the small shifts of positions. We generate $\Psi'[i]$ by iterating over $\Psi[j]$, where the indices might differ because we skip position q, so $j = i$ for $i < q$ and $j = i + 1$ for $i \geq q$.

1. If $\Psi[j] < q$, we set $\Psi'[i] = \Psi[j]$.
2. If $\Psi[j] > q$, we set $\Psi'[i] = \Psi[j] - 1$.

Both cases are just a simple copy of Ψ to Ψ' considering that we skip position q. If Ψ points to a position after q, we subtract one, because all such positions will be shifted once to the left.

3. If $\Psi[j] = q$ and $\Psi[\Psi[j]] < q$, we set $\Psi'[i] = \Psi[\Psi[j]]$
4. If $\Psi[j] = q$ and $\Psi[\Psi[j]] > q$, we set $\Psi'[i] = \Psi[\Psi[j]] - 1$

These two cases handle the position j with $\Psi[j] = q$, which is the position we want to skip. Obviously, only one of the two cases occurs per computation of Ψ', and this case

applies to only one position, so they are the last two cases tested due to the efficiency. We skip the value q, which is why we use $\Psi[\Psi[j]]$. In principle, the case distinction is the same as in the previous two cases: If the next position is higher, we need to subtract one for the shift. The latter two cases also answer the question, whether we delete the highest node of the edge, because we always get the case that $\Psi[\Psi[j]] > q$ unless v is the highest node.

In the example, we get

$$\Psi' = [2, 4, 7, 10, 8, 5, 6, 9, 11, 0, 3, 1]$$

from

$$\Psi = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1].$$

At position $q' = 4$ where $\Psi[q'] = q = 8$, we retrieve for $\Psi'[4]$ the value $\Psi[\Psi[4]] = 11$ with the fourth case from above. Because $11 \geq 8$, we decrease 11 by 1, so we get $\Psi'[4] = 10$. Also, all references to positions greater than 8 are reduced by 1. But this has the negative effect that the subsequence $7, 10, 8$, which belongs to node 1, is no longer a contiguous increasing subsequence. Therefore, we now explain the function for modifying Ψ' such that the returned Ψ'' fulfills the contiguous increasing subsequences property again. The following operations will be used by other operations like insertNode$(\cdot)$ as well. Also, these steps are the last steps necessary for completing removeNode$(\cdot)$.

4.3 Interval Upward and Downward Sort

Assume that we are given a new D', Ψ', a position p and the information that p is the only position that violates the contiguous increasing subsequence property. In this section, we will explain the algorithm for the case that $\Psi'[p] > \Psi'[p+1]$, which we call *interval upward sort*. The case $\Psi'[p] > \Psi'[p-1]$ works analogously and is called *interval downward sort*. From the modifications to the HyperCSA structure, we always know if we need the upward or downward sort algorithm. We know by $\Psi'[p] > \Psi'[p+1]$ that the value $\Psi'[p]$ should have a higher position within the interval $S(v)$ of the node v that belongs to position p. This is the reason we call this modification the *interval upward* sort.

In our example, we get $D' = 1010010001011$, $\Psi' = [2, 4, 7, 10, 8, 5, 6, 9, 11, 0, 3, 1]$, and $p = 3$. We also know that the subsequence $[7, 10, 8]$ is not sorted correctly and that the $10 = \Psi[3]$ needs to be moved to a higher position within the interval $S(1)$.

First, we find the position q to switch the value of $\Psi'[p]$ to, which is the position within $S(v)$ such that $\Psi'[q] < \Psi'[p] < \Psi'[q+1]$. We find the position q by scanning $S(v)$ starting at p upwards towards the end of $S(v)$. Note, that if the scanning results in $p = q$, then everything is already sorted correctly. Also, finding q can be done in at most $|S(v)| = \text{degree}(v)$ steps. In our example, we get $q = 4$.

Second, we move every value in $(p, q]$ to the next smaller position and move the value $\Psi'[p]$ to q. We call the newly updated function Ψ''. This sorts the interval $S(v)$ correctly. This step obviously also needs only degree(v) steps. However, it creates the problem that all positions z with $\Psi'[z] \in [p, q] \subseteq S(v)$ need to be updated, because we changed the order of these values. We fix this with the steps three and four.

In our example, $(p, q] = (3, 4]$, so we only swap the values at the positions 3 and 4. Thus, we get $\Psi' = [2, 4, 7, 8, 10, 5, 6, 9, 11, 0, 3, 1]$. This joins the two cycles of the edges edge(3) and edge(4), so we need to adjust the jumps to the swapped values.

Third, we fix the jumps $\Psi'[z]$ towards the values in $(p, q]$. Note that we shifted the intended destination of the jumps down by one position, so the values $\Psi'[z]$ need to be decreased by one. We find such a position z with $\Psi'[z] \in (p, q]$ by iterating over Ψ'' until $(\Psi'')^k[y] \in (p, q]$ for a $y \in [p, q)$. We know $k = \text{rank}(e)$ for the edge e that belongs to position y, because the cycle of the edge is correct except for the last jump at $z = (\Psi'')^{k-1}[y]$. This means, we get the position z in at most R_G steps. We then set $\Psi''[z] = \Psi'[z] - 1$. This small decrease in value cannot cause a new violation of the contiguous increasing subsequence property at the positions z, because all values in Ψ' are unique and, by a decrease by one and restoring the uniqueness in step four by only one adjustment, the increasing order of the values around z cannot have changed, as this would require a change of at least two. A consequence of the order at positions z with $\Psi''[z] \in [p, q)$ not causing a new violation is, that we can update the values and do not have any branches of additional necessary updates other than step four. Thereby, we only have at most degree(v) edges to update and need at most $\mathcal{O}(R_G)$ steps for such an update.

In our example, we only need to fix the jump to position 4, which is now at position 3. We therefore traverse from the value starting at $\Psi''[3]$ until we get to position z with $\Psi''[z] = 4$, which requires the lookups at 8, 11 and 1. Then, we set $\Psi''[1] = 4 - 1 = 3$, so we have $\Psi'' = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 3, 1]$. Note that we have a second 3 here, which we will update in the next step.

Fourth, for the value z_p with $\Psi'[z_p] = p$, we set $\Psi''[z_p] = q$. Like in step three, we use the partial cycle in Ψ'' containing q to get to position z_p and then set $\Psi''[z_p] = q$. Note that we increased the value from p to q here. This means that z_p may now be a position that violates the contiguous increasing subsequence property in the same way p did before our adjustment. Thus, we set $p = z_p$ and restart with the first step.

In our example, we get $z_p = 10$ with $\Psi''[z_p] = p = 3$ after just one step by traversing $\Psi''[q] = \Psi''[4] = 10$. Now, we set $\Psi''[z_p] = q = 4$ and get

$$\Psi'' = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 4, 1].$$

We would continue with $p = z_p$. We can end iterating here, because $[0, 4]$ is a contiguous increasing subsequence and we are finished with the interval upward sort.

It is not trivial to see how long we must sort intervals. First, we prove that we need at most $R_G \cdot \Delta_G$ update steps. Then, we will prove that this algorithm needs at most $2R_G$ such update steps in case of removing or inserting a node to an edge.

Theorem 1. *Interval upward sort terminates after at most $R_G \cdot \Delta_G$ iterations. Its total runtime is in $\mathcal{O}(R_G^2 \cdot \Delta_G^2)$.*

Proof. First, if e is the edge that belongs to position p of the first iteration, all subsequent values of p are also representatives of edge e. The first p can thereby only move to the end of the interval $S(v)$ and, in the worst-case, is only moved one step per visit of the interval $S(v)$. After at most R_G steps, we visit $S(v)$. Thus, after at most $R_G\Delta_G$ iterations, we aborted early or we have moved p to the end of $S(v)$.

The runtime of each iteration is dominated by the third step, which takes $\mathcal{O}(R_G \cdot \Delta_G)$ steps. As we have at most $R_G \cdot \Delta_G$ iterations, the worst-case time is $\mathcal{O}(R_G^2 \cdot \Delta_G^2)$. □

Theorem 2. *We get an upper bound of $2R_G$ iterations if the interval upward or downward sort is called by removing or adding a single node. In this case, the interval upwards or downwards sort's worst-case runtime is $\mathcal{O}(R_G^2 \cdot \Delta_G)$.*

We can get an upper bound of $2R_G$ iterations if the interval upward or downward sort is called by removing or adding a single node. Then, interval upwards or downwards sort's worst-case runtime is $\mathcal{O}(R_G^2 \cdot \Delta_G)$.

Proof. First, if e is the edge that belongs to position p of the first iteration, all subsequent values of p also represent the same edge e. Next, we consider the suffixes of T: The move from position p to q sorts e towards the edges that have at least the same first symbol of the suffix, due to the value of Ψ. So, after $\text{rank}(e)$ steps, we update p inductively correctly by all $\text{rank}(e)$ symbols of its suffix. The former suffix during the construction is no longer necessary to consider as we used it only to obtain the contiguous increasing subsequence property in the first place. Thus, the reference of p is moved to the correct position at step $\text{rank}(e)+1 \leq R+1$. The second round of updates is adjusting the numbers according to this now corrected position. The next visit of $S(v)$ does not require any further changes in the interval. □

The overall result of removeNode$(3, 2)$ is $\Psi'' = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 4, 1]$ and $D' = 1010010001011$.

4.4 Inserting a Node to an Edge

With the call insertNode(p, v), we add the node v to the edge edge(p). As an example, we will show insertNode$(4, 2)$ on the result of the removeNode$(3, 2)$ example, so we have $D = 1010010001011$ and $\Psi = [2, 3, 7, 8, 10, 5, 6, 9, 11, 0, 4, 1]$ initially. The call insertNode$(4, 2)$ will revert the change done by removeNode$(3, 2)$, so the result will be $D' = 10100100001011$ and $\Psi'' = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1]$. We highlight that the representative 3 for the removal and the representative 4 for the insertion represent the same edge, but the actual value was changed due to the modifications made during the removal in the last section.

First, we need to find the pair of positions (i, j) such that $\Psi(i) = j$ and $\text{node}_D(i) < v < \text{node}_D(j)$. If such a pair does not exist, because for all nodes $v' \in \text{edge}(p)$, we have either $v' < v$ or $v' > v$, we want the pair (i, j) such that $\Psi(i) = j$ and $\text{node}_D(i) > \text{node}_D(j)$. During this traversal, if we already have $v \in \text{edge}(p)$, we abort the insertion early.

The next major goal is to insert j into the subsequence $\Psi[S(v)]$. Therefore, we first search the position $q \in S(v) \cup \{\max(S(v)) + 1\}$ with $\Psi[q-1] < j < \Psi[q]$ (to simplify, we assume $\Psi[-1] = -1$ and $\Psi[M_G] = M_G$ for the boundaries here). Because $\Psi[S(v)]$ is a contiguous increasing subsequence, we can use a binary search to find q.

In the example, we have $(i, j) = (4, 10)$. We search the position q in the subsequence $[5, 6, 9, 11]$ where we should put the value 10 using a binary search. We want to insert the 10 between 9 and 11, so $q = 8$ in Ψ.

Second, we copy D to D', but at the v-th span of D, we increase the length of that span by one. In the example, we get $D' = 10100100001011$.

Third, we do a similar step for Ψ': If both $z, \Psi[z] < q$, we set $\Psi'[z] = \Psi[z]$. If $z < q$ and $\Psi[z] \geq q$, we set $\Psi'[z] = \Psi[z] + 1$. If $z \geq q$ and $\Psi[z] < q$, we set $\Psi'[z+1] = \Psi[z]$. If both $z, \Psi[z] \geq q$, we set $\Psi'[z+1] = \Psi[z] + 1$. In other words, we move values higher than q up by one to create the space we need for the new node in the edge. We can now set $\Psi'[i] = q$ and $\Psi'[q] = j + 1$ and thereby inserted the node v to the edge.

In the example, the first copy steps results in $[2, 3, 7, 9, 11, 5, 6, 10, , 12, 0, 4, 1]$ and the two explicit assignments produce $\Psi' = [2, 3, 7, 9, 8, 5, 6, 10, 11, 12, 0, 4, 1]$. Now, the subsequence $[7, 9, 8]$ of node 1 is not ordered properly.

By the binary search, we know that the subsequence of the interval $S(v)$ is properly ordered. In the same way as for deleting a node from an edge, the increase by one cannot violate the contiguous increasing subsequence property, so the only position that might be incorrect is position i. Again, we can use the interval downward or upward sort here. We need the interval downward sort if we reduced $\Psi[i]$, as in our example, or the interval upward sort, if we increased $\Psi[i]$. Increasing $\Psi[i]$ happens if and only if we add a new highest node to the edge. In the example, we get $\Psi'' = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1]$ as the result after the interval downward sort.

The runtime of insertNode(p, v) is dominated by the steps of copying D to D' and Ψ to Ψ', which both take $\mathcal{O}(M_G)$ time, or by the interval upward or downward sort using $\mathcal{O}(R_G^2 \cdot \Delta_G)$ time. So, in total we get $\mathcal{O}(M_G + R_G^2 \cdot \Delta_G)$ in the worst-case.

4.5 Inserting a new Edge

The call addEdge(e) inserts a new edge e into the existing hypergraph G. Compared to the previous three modifications of the structure, we do not have a representative p to work with. We assume that the nodes in e are sorted, so $e[0] < e[1] < \cdots < e[\text{rank}(e)-1]$. As an example, we will insert the edge $\{1, 2, 3\}$ into the hypergraph with the edges $\{2\}$, $\{2\}$, $\{0, 1, 2, 3\}$, and $\{0, 1, 2, 4\}$, which is given by $\Psi = [2, 3, 6, 7, 4, 5, 8, 9, 0, 1]$ and $D = 10101000111$. This will undo the step of deleteEdge(3) in Section 4.1.

For e, we need one new position for every node $e[i]$. More precisely, for $i < \text{rank}(e)$, we insert one new element in every subsequence in $\Psi[S(e[i])]$. Using this information about the location of the newly inserted elements, we compute D' from D as follows: Let D' be a bitvector of length of $M_G + 1 + \text{rank}(e)$. We iterate over D and set the current position of D' to the bit occurring at $D[i]$ and we skip one position of D' if $D[i] = 1$ and $\text{node}_D(i) \in e$. Because the nodes in e are ordered, we will find the $e[i]$ in order of their index i. This results in $D' = 10100100001011$ from $D = 10101000111$, where we increased the length of the spans associated with the nodes 1, 2, and 3.

The HyperCSA structure was constructed using suffix sorting. To get the insert positions for the nodes of e, we will use this construction idea. The problem is that we modified Ψ_{CSA} to Ψ: From any position p in Ψ, we only know a prefix of the suffixes of the construction string T. For example, for any edge e', the longest exactly known prefix of a suffix starts at $e'[0]$. On the contrary, we know the construction rules of T, so we know that after edge e' was an edge e'' with $e' > e''$ or e' was the lexicographically smallest edge.

Our goal is to find the insert position p_0 of $e[0]$ in $S(e[0])$. We will use a binary search on the interval $S(e[0])$ and compare e to the current position. This position might not be a first node of an edge: In our example, the interval $S(e[0]) = S(1) = [2, 3]$ contains representatives to the edges $\{0, 1, 2, 3\}$, and $\{0, 1, 2, 4\}$, which both do not have the node 1 as their first node. Thus, we always compare the edge e to a part s of another edge e' that contains the node $e[0]$.

Let e' be another edge that contains $e[0]$. We only consider the part s of e' that consists exactly of the nodes of e' larger or equal to node $e[0]$. Therefore, we call s an *edge-suffix*. We refer with $\text{rank}(s)$ to the length of s. We end up in one of several cases:

- If $e[m] < s[m]$ for an $m < \min(\text{rank}(e), \text{rank}(s))$ and $e[i] = s[i]$ for all $i < m$: e has a smaller suffix than s.
- If $e[m] > s[m]$ for an $m < \min(\text{rank}(e), \text{rank}(s))$ and $e[i] = s[i]$ for all $i < m$: e has a larger suffix than s.

– If $e[i] = s[i]$ for all $i < \min(\text{rank}(e), \text{rank}(s))$:
  - If $\text{rank}(e) < \text{rank}(s)$: e has a smaller suffix than s.
  - If $\text{rank}(e) > \text{rank}(s)$: e has a higher suffix than s.
  - If $\text{rank}(e) = \text{rank}(s)$:
    * $e \neq e'$: e has a higher suffix than e'.
    * $e = e'$: we can choose, because both decisions will give us the same result.

The first two cases are the classic suffix sorting cases, so the first unequal node determines the order of the suffixes. Next, if all nodes are equal, one suffix can be longer than the other, which is the case if the ranks of e and s are different. Let us consider the case $\text{rank}(s) > \text{rank}(e)$ and $e[i] = s[i]$ for all $i < \text{rank}(e)$: The next node in s is larger than the previous and the next node n after the last node of e is smaller or equal to its previous node due to the construction of T, so $s[\text{rank}(e)] > s[\text{rank}(e)-1] = e[\text{rank}(e) - 1] \geq n$. Thus, e has the smaller suffix. The case $\text{rank}(s) < \text{rank}(e)$ works analogously. Here, the special case is if $\text{rank}(e) = \text{rank}(s)$, so we have $e = s$. If $s \neq e'$, there is a node $n \in e'$ that is strictly smaller than any node in e and we get $e > e'$. In T, the suffix after e is higher than the suffix after e' by the downward sort of the edges and thereby we have that e has the higher suffix. If $e' = s$, then $e' = e$. This means, their cycles would be next to each other in Ψ. In this rare case, we can use the cycle of e' to get all positions of e' easily, either all are before or all after the cycle of e'. This results in the same Ψ in both cases.

In the example of inserting $e = \{1, 2, 3\}$, the interval is $S(e[0]) = S(1) = [2, 3]$ and we first compare e to the edge containing position 3. Because $\Psi[3] = 7 \in S(2)$, we also have $e[1] = s[1]$. Next, $\Psi[7] = 9 \in S(4)$, so we have $e[2] = 3 < 4 = s[2]$, so e has the smaller suffix. By the binary search, we now compare to $\Psi[2] = 6$. As $6 \in S(2)$ and $\Psi[6] = 8 \in S(3)$, we did $\text{rank}(e)$ comparisons. Because $\Psi[8] = 0 < 8$, we also have reached the end of the edge-suffix s here, so $\text{rank}(e) = \text{rank}(s)$. If $e' = s$, then we should have $\Psi[8] \in S(1)$, but we have $\Psi[8] = 0 < S(1)$, so $e > s$. Thereby, the insert position is 3, which means we insert e before the edge e at position $p_0 = 3$.

Using the S-type position to determine the end of the edge-suffix s and the fact that the implication $e' \neq s \Rightarrow e' \neq e$ requires $e = s$, we can efficiently obtain the position of $e[0]$ without further extension of the cases above. To conclude, we can decide within at most R_G steps if e or s has the higher suffix and we need at most $\mathcal{O}(\log(\Delta_G))$ steps for the binary sort, so we can get the insert position p_0 of $e[0]$ in $\mathcal{O}(R_G \cdot \log(\Delta_G))$ steps.

For the other nodes, we use the search from insertNode($\cdot$): We know the position p_0 of $e[0]$, so we can search in the subsequence $\Psi[S(e[\text{rank}(e)-1])]$ for position q with the smallest value higher than or equal to p_0 using a binary search. We set $p_{\text{rank}(e)-1} = q$, because we want the contiguous increasing subsequence property. $\Psi[S(e[\text{rank}(e)-1])]$ fulfills the property, and we will set $\Psi'[p_{\text{rank}(e)-1}] = p_0$. Having $p_{\text{rank}(e)-1}$, we can search for $p_{\text{rank}(e)-2}$. So we iterate downward to search the needed positions until we find p_1. For the purpose of storing and using the information best, we use a bitvector Z of length $|D| = M_G + 1$ that is initially 0, and we set $Z[p_i] = 1$ for the found insert positions p_i for all $i < \text{rank}(e)$.

In our example, we saw that $p_0 = 3$. As $\text{rank}(e) = 3$, we next search the position p_2. Thus, we search in Ψ in the interval $S(e[2]) = S(3) = [8, 9)$ for the value $p_0 = 3$ using a binary search. Because $\psi[8] = 0$, we get $p_2 = 9$. Thereafter, we continue for p_1: Here, $S(2) = [4, 8)$ and we get $p_1 = 7$, because $\Psi[6] = 8$, $\Psi[7] = 9$, and we searched for the value $p_2 = 9$. To conclude, we do not get the values p_i, but we get $Z = 00010001010$.

Given Z, the construction of Ψ' is straightforward: We iterate over Ψ with index i, and if at the current position i the value $Z[i]$ is 1, we skip a position in Ψ', so the index j of Ψ' is increased by an additional step. This produces the yet unassigned positions in Ψ' for the new edge e. Additionally, we need to adjust the value $\Psi[i]$ before we can set it to $\Psi'[j]$. We must increase $\Psi[i]$ by the number of added unassigned positions up to $\Psi[i]$, so we set $\Psi'[j] = \Psi[i] + \text{rank}_Z(\Psi[i] + 1, 1)$, where the 1s in Z are exactly the number of unassigned positions. We increment $\Psi[i]$ by one, because if $Z[i] = 1$, we also need to consider that $Z[i] = 1$ means shifting this character by an additional position for a new unassigned position. We remark that we use a data structure to answer the rank queries in $\mathcal{O}(1)$ time, which highlights the advantage of encoding all insert positions p_i by a single bitvector Z.

In the example, we use the bitvector $Z = 00010001010$ to transform

$$\Psi = [2, 3, 6, 7, 4, 5, 8, 9, 0, 1]$$

into

$$\Psi = [2, 4, 7, _, 9, 5, 6, 10, _, 12, 0, _, 1].$$

The underscores stand for the unassigned insert positions.

As the last step of inserting edge e, we assign values to the unassigned positions of the new edge. We use $p_{i-1} = \text{select}_Z(i, 1) + i$ for $1 \leq i \leq \text{rank}(e)$ to get the position of node $e[i-1]$. So, we set $\Psi'[p_{i-1}] = i_i$ for all $0 \leq i < \text{rank}(e)$ and $\Psi'[p_{\text{rank}(e-1)}] = p_0$.

In the example, we get $\Psi' = [2, 4, 7, 8, 9, 5, 6, 10, 11, 12, 0, 3, 1]$. Together with $D' = 10100100001011$, we correctly inserted $e = \{1, 2, 3\}$ and reverted the changes.

Overall, for any edge $e \subseteq V$, we have

$$(\Psi', D') = \text{addEdge}(\Psi, D, e)$$

and

$$\text{deleteEdge}(\Psi', D', \text{exists}(\Psi', D', e)[0]) = (\Psi, D),$$

so we can revert any addition of a new edge. Also, if $e = \text{edge}(p)$, we have for any representative p,

$$\Psi', D' = \text{deleteEdge}(\Psi, D, p),$$

and

$$(\Psi, D) = \text{addEdge}(\Psi', D', e),$$

so the deletion is invertible if and only if we are allowed to delete the edge that belongs to p. We extend this by a special call $\text{exists}(\Psi, D, e)$, which gives us a set of edge representatives of edges that are equal to the edge represented by p. We remark that we are not allowed to delete an edge if we thereby delete the last connection of a node. In the next paragraph, we present an idea to overcome this issue.

4.6 Unconnected Nodes in HyperCSA

With deleteEdge($\cdot$) or removeNode($\cdot$), we can reach the case that we want to delete the last connection of a node n. In this chapter, we present an easy solution to that problem. Let N be a bitvector of length $|V_G|$. If the node $n \in V$ has at least one incident hyperedge e, then $N[n] = 1$. Up to this point, this was a requirement, but

we allow nodes n that are not connected to any hyperedge by setting $N[n] = 0$. Consequently, we can obtain the node degrees as

$$\text{degree}(n) = \begin{cases} 0 & N[n] = 0 \\ |S(\text{rank}_N(n,1))| & N[n] = 1 \end{cases}.$$

Thus, we can now delete the last connection of a node n by setting $N[n] = 0$. This occurs while deleting a span from D that had only length 1, so if we had $D[i] = D[i+1] = 1$ for $i = \text{select}_D(n, 1)$.

Allowing unconnected nodes needs a small adjustment for queries and for modifications of HyperCSA. First, we look at the queries: Given a sequence s of nodes n_i for any query, we check for every node n_i in s that $N[n_i] = 1$, otherwise the query obviously has no matching edge by definition of N. If all nodes are connected, we need to translate them, because there might be lower nodes that are not connected to any edge. We set $n'_i = \text{rank}_N(n_i, 1)$ and s' to be the sequence of the n'_i. The rank$(\cdot)$ reduces n_i by the number of zeros in N before n_i. We can then perform the query using s'. Without the translation, we search within the wrong spans of D.

We adjust the values of nodes using the transformation from n_i to n'_i of the previous paragraph. In the case of adding at least one node n to the structure that has $N[n] = 0$, we set $N[n] = 1$ and add a new span of length 1 to D at the right location besides the changes to Ψ described in Section 4.4 or Section 4.5.

It is now also possible to add a new node to G by appending a zero to N. If the node should be inserted in between other nodes, we shift the values of higher positions in N to the next higher position. We delete a node n from G that has $N[n] = 0$, by deleting the n-th bit in N, so we shift the bits with a position higher than n to the position one smaller and shorten the length of N by one. If we want to delete a node with $N[n] = 1$, we first detach all connected edges from that node. A naive solution calls removeNode$(\text{select}_D(n, 1), n)$ for $S(n)$ times. Each such call will detach one edge from the node n. This method is naive, because we could detach all edges from the node n at the same time and reduce the update cost for the remaining parts of D and Ψ, which dominates the runtime. We do not present an advanced algorithm here, because in our opinion, deleting a connected node happens rarely in real-world scenarios.

5 Experimental Evaluation

Our tests were performed on a Debian 5.10.209-2 machine with 512GB RAM and 128 Cores Intel(R) Xeon(R) Platinum 8462Y+ @ 2.80GHz. We evaluated the social network community hypergraphs CoFr and CoOr, which are available from SNAP [5]. The statistics of the datasets are given in Table 2.

hypergraph G	$\lvert V_G\rvert$	$\lvert E_G\rvert$	M_G	R_G	Δ_G
com-friendster (CoFr)	7,944,949	1,620,991	23,479,217	9,299	1,700
com-orkut (CoOr)	2,322,299	15,301,901	107,080,530	9,120	2,958

Table 2: Used datasets from SNAP [5].

Construction of the CoFr dataset takes 3 minutes and 33.951 seconds. Deleting an edge from CoFr in the HyperCSA format takes only about 26.8349 seconds. We added

an edge within about 27.1503 seconds. Inserting a node takes about 38.0888 seconds. Attempting to delete a node from an edge to which the node is not connected takes only 7.366 milliseconds.

The construction of the CoOr dataset takes about 51 minutes. Deleting an edge takes about 93.056 seconds. We add a new edge to the HyperCSA-compressed CoOr hypergraph in about 93.2257 seconds. Attempting to delete a node from an edge to which the node is not connected takes only 17.381 milliseconds, whereas adding a new node to an edge takes about 126.184 seconds.

The results of both CoFr and CoOr show that our update-algorithms are significantly faster than a decompress-update-compress approach, which needs the time to compress the datasets and the updates have only a negligible influence on the compression time. If multiple updates are needed, for example deleting multiple edges at a time, these steps can be combined to reduce the overall time needed.

6 Conclusion

In this paper, we introduced algorithms to add and remove nodes or edges and algorithms to add or remove single nodes from edges in a hypergraph that was compressed using HyperCSA. We provided worst-case runtimes for all presented algorithms and showed by a practical evaluation that the algorithms are much faster than the naive decompress-update-compress approach.

References

1. E. Adler, S. Böttcher, and R. Hartel: *Compressing hypergraphs using suffix sorting*, in Data Compression Conference, DCC 2026, Snowbird, UT, USA, March 24-27, 2026, IEEE, 2026, pp. 113–122.
2. N. R. Brisaboa, A. Cerdeira-Pena, A. Fariña, and G. Navarro: *A Compact RDF Store Using Suffix Arrays*, in String Processing and Information Retrieval, Cham, 2015, Springer International Publishing, pp. 103–115.
3. P. Elias: *Universal codeword sets and representations of the integers.* IEEE Transactions on Information Theory, 21(2) 1975, pp. 194–203.
4. G. Ghoshal, V. Zlatic, G. Caldarelli, and M. E. J. Newman: *Random hypergraphs and their applications.* CoRR, abs/0903.0419 2009.
5. J. Leskovec and A. Krevl: *SNAP Datasets: Stanford Large Network Dataset Collection.* http://snap.stanford.edu/data, June 2014, Last Accessed: 19.04.2026.
6. L. Li and T. Li: *News recommendation via hypergraph learning: encapsulation of user behavior and news content*, in Proceedings of the Sixth ACM International Conference on Web Search and Data Mining, WSDM '13, New York, NY, USA, 2013, Association for Computing Machinery, p. 305–314.
7. Y. Liu, Q. Luo, M. Xiao, D. Yu, H. Chen, and X. Cheng: *Reordering and compression for hypergraph processing.* IEEE Trans. Computers, 73(6) 2024, pp. 1486–1499.
8. U. Manber and G. Myers: *Suffix Arrays: A New Method for On-Line String Searches*, in Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA, 1990, pp. 319–327.
9. K. Sadakane: *New text indexing functionalities of the compressed suffix arrays.* J. Algorithms, 48(2) 2003, pp. 294–313.
10. J. Shun: *Practical parallel hypergraph algorithms*, in PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020, 2020, pp. 232–249.
11. S. Tan, J. Bu, C. Chen, B. Xu, C. Wang, and X. He: *Using rich social media information for music recommendation via hypergraph model.* ACM Trans. Multimedia Comput. Commun. Appl., 7S(1) Nov. 2011.

12. S. Tan, Z. Guan, D. Cai, X. Qin, J. Bu, and C. Chen: *Mapping users across networks by manifold alignment on hypergraph*, in Proceedings of the AAAI Conference on Artificial Intelligence, vol. 28, 2014.
13. T. Zhao, D. Zhao, L. Guo, and N. R. Tallent: *Hybhuff: Lossless compression for hypergraphs via entropy-guided huffman-bitwise coordination.* CoRR, abs/2506.15844 2025.

Author Index

Adler, Enno, 97

Böttcher, Stefan, 97
Bannai, Hideo, 13

Fujimaru, Hiroto, 43

Hartel, Rita, 97
Hasibi, Hamed, 28

Inenaga, Shunsuke, 13, 43, 58

Köppl, Dominik, 58

Lecroq, Thierry, 85
Lefebvre, Arnaud, 3, 85
Leonard, Laurentius, 13
Louza, Felipe A., 3

Marino, Francesco Pio, 85
Mhaskar, Neerja, 28
Mieno, Takuya, 13, 58

Shibata, Hiroki, 58
Skulimowski, Stanisław Piotr, 71
Steiner, Teresa Anna, 1

Zhang, Tieyun, 28

Proceedings of the Prague Stringology Conference 2026
Edited by Jan Holub and Jan Žďárek
Published by: Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science
Prague Stringology Club
Thákurova 9, Praha 6, 160 00, Czech Republic.
122 pages, first edition.

ISBN 978-80-01-07576-0

URL: http://www.stringology.org/
E-mail: psc@stringology.org Phone: +420-2-2435-9811

Printed by powerprint s.r.o.
Brandejsovo nám. 1219/1, Praha 6 Suchdol, 165 00, Czech Republic

© Czech Technical University in Prague, Czech Republic, 2026