CLOSED STRINGS

EFFICIENT COMPUTATION OF CLOSED SUBSTRINGS

By SAMKITH K JAIN, B.Eng.

A Thesis Submitted to the School of Graduate Studies in Partial Fulfillment of the Requirements for the Degree Master of Science

McMaster University

MASTER OF SCIENCE (2025)

Hamilton, Ontario, Canada (Computing and Software)

TITLE:	Efficient Computation of Closed Substrings
AUTHOR:	Samkith K Jain B.Eng. (Computer Science and Engineering), Visvesvaraya Technological University, Belagavi, Karnataka
SUPERVISOR:	Dr. Neerja Mhaskar
NUMBER OF PAGES:	xi, 59

Lay Abstract

Closed strings are unique patterns found in sequences such as DNA or binary data, exhibiting interesting properties with both theoretical and practical significance. This work presents an efficient method for identifying all closed substrings within a sequence, even for very large inputs. By focusing on Fibonacci words, predictable patterns in the number of maximal closed substrings were discovered. These findings improve our understanding of the structure and combinatorial characteristics of such sequences.

Abstract

A ***closed string*** u is either of length one or contains a border that occurs only as a prefix and as a suffix of u, without appearing elsewhere within u. This thesis presents a fast and practical $\mathcal{O}(n \log n)$ time algorithm to compute all $\mathcal{O}(n^2)$ closed substrings of a string $w[1..n]$. This is achieved by introducing a compact representation of all closed substrings using only $\mathcal{O}(n \log n)$ space. Additionally, a simple and space-efficient approach is proposed to compute all maximal closed substrings (MCSs) using the suffix array (SA) and the longest common prefix (LCP) array of $w[1..n]$.

Given a Fibonacci word f_n, where $n > 5$, the thesis shows that the exact number of MCSs is $M(f_n) \approx \left(1 + \frac{1}{\phi^2}\right) F_n \approx 1.382 F_n$, where ϕ is the golden ratio, and F_n is the n-th Fibonacci number. These results highlight novel combinatorial properties of closed substrings in structured sequences.

To complement the theoretical findings, an efficient implementation of the algorithms for computing closed strings and MCSs is provided. The implementation has been thoroughly tested and is designed to support practical applications, facilitating further exploration of closed substrings in various contexts.

For the love of Computer Science, and all who are inspired by it.

Acknowledgments

I extend my heartfelt gratitude to my advisor, Dr. Neerja Mhaskar, for her unwavering guidance throughout this journey. Her invaluable feedback and support greatly enriched my critical thinking and deepened my understanding, ultimately enabling me to achieve new results.

I sincerely thank my examiners, Dr. Antoine Deza and Dr. Ryszard Janicki, for their valuable feedback and critical evaluation, which helped refine my work and contributed significantly to the quality of this research.

I also thank the professors, administrative staff, and support staff of the Computing and Software department for creating an environment that made my master's experience both enriching and seamless.

Table of Contents

Lay Abstract iii

Abstract iv

Acknowledgments vi

Abbreviations xi

Declaration of Academic Achievement xii

1 Introduction 1

1.1 Motivation 2

1.2 Contribution 2

1.3 Overview 3

2 Preliminaries 5

2.1 Definitions 5

2.2 Arrays 7

2.3 Suffix Tree 9

2.4 The Extended Dis-joint Set 10

3 Related Work **14**

3.1 Closed Strings 14

3.2 Covers 18

3.3 Gapped-Repeats 19

4 Closed Substrings **21**

4.1 Naive Computation of All Closed Substrings 21

4.2 Compact Representation of Closed Substrings 22

4.3 Maximal Right-Closed $\mathcal{MRC}$ Array 25

4.4 $\mathcal{MRC}$ Array using Suffix Trees 26

4.5 $\mathcal{MRC}$ Array using SA and LCP 32

4.6 Algorithm to Compute $\mathcal{C}(w)$ 36

4.7 Computing All MCSs 37

5 MCSs in Fibonacci Words **38**

6 Implementation and Experiments **45**

6.1 Experimental Setup 45

6.2 Key Data Structures 46

6.3 String Generation 47

6.4 Results 48

7 Conclusion and Open Problems **50**

7.1 Conclusion 50

7.2 Open Problems 51

List of Figures

2.1 An example of a suffix tree ($\mathcal{T}_w$) for a string $w[1..12] = mississippi\$$. 11

4.1 Illustration of the second case in the proof of Lemma 2, showing that $|p_j|$ can be computed from $|b_j|$, $|b_{j-1}|$ and $|r_j|$, such that $p_j = r_j$ or $p_j \in E_{r_j}$. Note that when $p_j = r_j$, $p'_j = \varepsilon$ [29]. 24

4.2 Suffix tree $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$. Internal nodes are processed in bottom-up order from A to F, each internal node is annotated with its computed *ChangeList*, which identifies the maximal right-closed substrings. 30

6.1 Maximum number of MCSs in strings of length n, where $1 \leq n \leq 20$, for alphabets of size $\sigma = 2, 3$ and 4 [29]. 49

List of Tables

2.1 SA and LCP for the string $w[1..11] = mississippi$, the suffixes of string w are lexicographically sorted. 8

2.2 The border array β of string $w[1..11] = abracadabra$. 9

2.3 The open-closed sequence OC of string $w[1..11] = abracadabra$. 9

4.1 Naive computation of all closed substrings of $w[1..11] = mississippi$. 22

4.2 Compact representation $\mathcal{C}(w)$ of $w[1..11] = mississippi$ with corresponding closed factors, maximal right-closed factors, and MCSs [29]. 26

4.3 The $\mathcal{MRC}$ array for the string $w[1..11] = mississippi$ [29]. 27

4.4 Path-labels and string-depths of all internal nodes of $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$. 29

4.5 Tracing of the next$[i]$ array for each internal node of $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$ during the computation of its $\mathcal{MRC}$ array. . 31

4.6 *ChangeList* computation at each internal node of $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$. 31

5.1 f_n and F_n, for $0 \leq n \leq 8$ 38

5.2 p_n and P_n, for $0 \leq n \leq 8$ 39

Abbreviations

MCS	Maximal Closed Substring
SA	Suffix Array
LCP	Longest Common Prefix Array
OC	Open-Closed Sequence or Array
AVL Tree	Adelson-Velsky and Landis Tree

Declaration of Academic Achievement

I, Samkith K Jain, hereby declare that this thesis titled *Efficient Computation of Closed Substrings* is the result of my independent research work. The research described in this thesis was carried out solely by me under the supervision of Dr. Neerja Mhaskar. Contributions from colleagues, collaborators, or others are duly acknowledged in the Acknowledgments section of this thesis.

Specifically, I affirm that:

1. I identified and formulated the research problem and independently developed the algorithms presented in this thesis.

2. The literature review and background research included in this thesis were conducted by me.

3. Any external contributions, including technical support or discussions, are explicitly credited and acknowledged within the thesis.

Samkith K Jain

Chapter 1

Introduction

An ***alphabet*** Σ, of size σ, is a totally ordered set of symbols. A ***string (word)***, denoted by $w[1..n]$, is an element of Σ^*, whose length is n. The set Σ^* represents all possible strings (including the ***empty string*** ε) that can be formed using symbols from Σ. Specifically, $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ where Σ^k is the set of strings of length k over Σ. A ***substring (factor)*** $w[i..j]$ of w is a string, where $1 \leq i, j \leq n$, if $j > i$ then the substring is ε. A substring $w[i..j]$ is said to be a ***repeating substring*** if there exists another substring $w[i'..j']$ such that $i' \neq i$, $1 \leq i', j' \leq n$, and $w[i..j] = w[i'..j']$.

A ***Fibonacci*** word f_n is defined recursively by $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1}f_{n-2}$ for $n \geq 2$. We denote $F_n = |f_n|$, where F_n satisfies the Fibonacci recurrence relation $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 1$ and $F_1 = 1$.

The processing of strings to identify interesting patterns has a range of applications in a variety of different fields. Within the realm of computer science, patterns in strings have been helpful in lexical analysis of languages, finite automata, data compression, and much more. Other fields of study such as molecular biology, bioinformatics, cryptology, and geometry use string algorithms to generate insight into characteristic properties [43]. A wide range of string problems—including combinatorial puzzles, pattern matching, regularities in words, text compression, and miscellaneous challenges—are described in [20], along with efficient data structures that help solve these problems.

1.1 Motivation

Closed substrings (see Section 2.1 for definition) serve as structural boundaries within strings, marking points of symmetry and repetition that are critical to understanding the organization of sequences. They offer valuable insights into the combinatorial structure of specialized classes of strings, such as Sturmian and Trapezoidal words (see [22] for more details), which have deep connections to theoretical computer science and discrete mathematics. The study of these substrings is not purely theoretical; having a practical algorithm to compute and analyze closed substrings enables direct exploration of their properties. Such an implementation bridges the gap between theory and practice, providing a means to test conjectures, uncover new patterns, and address challenges in related fields. This dual approach of theory and practice makes closed substrings a compelling area of study with potential applications and discoveries.

1.2 Contribution

This thesis makes the following contributions:

- **Compact Representation of All Closed Substrings:** A compact data structure representing all $\mathcal{O}(n^2)$ closed substrings of a string $w[1..n]$ using only

$\mathcal{O}(n \log n)$ space.

- **Computation of All Closed Substrings:** A $\mathcal{O}(n \log n)$ time algorithm to find all closed substrings of w using SA and LCP in a compact representation, (see Sections 2.2.1 and 2.2.2, respectively, for definitions).

- **Computation of All MCSs (see Section 2.1 for definition):** A $\mathcal{O}(n \log n)$ time method to compute all MCSs of w using SA and LCP.

- **MCSs in Fibonacci Words:** A formula for the exact number of MCSs in Fibonacci words f_n, denoted by $M(f_n)$, and shows that $M(f_n) \approx 1.382F_n$, where f_n and F_n are the n-th Fibonacci word and number, respectively.

- The first implementation of algorithms to compute closed substrings and MCSs, facilitating practical experimentation and further research.

The results of this thesis have been accepted for the International Symposium on String Processing and Information Retrieval (SPIRE) 2025. The publication is available in Springer [29], and the code is publicly available at `https://github.com/neerjamhaskar/closedStrings`.

1.3 Overview

In Chapter 2, the key terminology and data structures used throughout this thesis are introduced. Chapter 3 summarizes related work on closed strings and related topics to provide background for this thesis. Chapter 4 defines a compact representation for all closed substrings of a string $w[1..n]$ and presents algorithms for computing all closed substrings and MCSs in $\mathcal{O}(n \log n)$ time. Chapter 5 explores the properties of

Fibonacci words and provides a formula to compute the number of MCSs in a given Fibonacci string f_n. Chapter 6 discusses the implementation details and experimental results. Finally, Chapter 7 concludes the thesis by summarizing the main findings and presents open problems for future research.

Chapter 2

Preliminaries

2.1 Definitions

An ***alphabet*** Σ, of size σ, is a totally ordered set of symbols. A ***string (word)***, denoted by $w[1..n]$, is an element of Σ^*, whose length is $n = |w|$. The set Σ^* represents all possible strings (including the ***empty string*** ε) that can be formed using symbols from Σ. For example, the string $w[1..11] = abracadabra$ has a length of $n = 11$. A ***substring (factor)*** $u = w[i..j]$ of w is a string where $1 \leq i, j \leq n$. If $j < i$, then the substring is ε. For example, $u = w[2..6] = braca$ is a substring of string w. The substring $w[i..j]$ is called a ***proper substring*** if $j - i + 1 < n$. For example, $u = w[2..5] = brac$ is a proper substring of w, whereas $u = w[1..11] = abracadabra$ is not proper because its length equals n.

A substring $w[i..j]$ where $i = 1$ is called a ***prefix*** of w. A ***proper prefix*** is a prefix of w that is shorter than w, that is, its length is less than n. For example, $w[1..4] = abra$ is a proper prefix of w. Likewise, a substring $w[i..j]$ where $j = n$ is called a ***suffix*** of w. A ***proper suffix*** is a suffix of w that is shorter than w, that is, its length is less than n. For example, $w[7..11] = dabra$ is a proper suffix of w.

If a substring u of $w[1..n]$ occurs as both a proper prefix and a proper suffix, then it is called a ***border*** of w. For example, in the string $w[1..11] = abracadabra$, the substring $u = abra$ is a border because $w[1..4] = w[8..11] = abra$ is a proper prefix and a proper suffix, respectively. Note that a string can have multiple borders, $u = a$ is also a border of w.

A ***closed string*** u is either of length one or contains a border that occurs only as a prefix and as a suffix in u and nowhere else within u; otherwise, it is called ***open***. In other words, the longest (possibly overlapping) border of a closed substring u does not have internal occurrences in u. For example, in the string $abcab$, ab occurs only as a prefix and a suffix. An occurrence of a substring u of a string w is said to be ***maximal right (left)-closed*** if it is a closed string and cannot be extended to the right (left) by a character to form another closed string. A ***maximal closed substring (MCS)*** of a string w is a substring that is both maximal left and maximal right-closed. An MCS of length one is called a ***singleton*** MCS; otherwise, it is called a ***non-singleton*** MCS. For example, in string $w[1..8] = abaccaba$, the non-singleton MCSs are $w[1..8] = abaccaba$, $w[3..6] = acca$, $w[1..3] = w[6..8] = aba$, $w[4..5] = cc$, and the singleton MCSs are all occurrences of a and b. Observe that $w[2..7] = baccab$, $w[2..8] = baccaba$, $w[1..7] = abaccab$, and both occurrences of c are closed but not maximal.

A ***cover*** of a string w is a substring u of w such that w can be constructed by overlapping and/or adjacent instances of u. If $|u| < |w|$, then it is said to be a ***proper cover***. For example, the string $w = abaababa$ has a proper cover aba (for more details on covers, see the survey [36]).

A string $w[1..n]$ is ***periodic*** with period t' if $w[i] = w[i + t']$ holds for all $1 \leq i \leq n - t'$. The ***shortest period*** of w is the smallest positive integer $t \leq t'$ for which this condition holds. A periodic substring $w[i..j]$ of string $w[1..n]$ with shortest period t is called a ***run*** if its length $j - i + 1 \geq 2t$ and its periodicity cannot be extended to the left or right, that is, $i = 1$ or $w[i-1] \neq w[i+t-1]$, and $j = n$ or $w[j+1] \neq w[j-t+1]$. For example, $w = mississippi$ has a run $u = ississi = (iss)^2 i$, where $t = 3$.

A ***gapped repeat*** is a substring of $w[1..n]$ of the form uvu, where u and v are non-empty strings. The substring u is called the ***arm*** of the gapped repeat. The period[1] of the gapped repeat is $|u| + |v|$. An occurrence of a gapped repeat is called a ***maximal gapped repeat*** if it cannot be extended to the left or to the right by at least one symbol while preserving its period. A maximal gapped repeat, which is also closed, is referred to as a ***gapped-MCS***. For example, in string $w = abacaba$, the gapped repeat is $abacab$, where $u = ab$ and $v = ac$ with period $|u| + |v| = 4$. The maximal gapped repeat is $abacaba$, as it cannot be extended further while preserving the period. Finally, the gapped-MCS is also $abacaba$, as $u = aba$ occurs exactly twice in it.

2.2 Arrays

2.2.1 Suffix Array (SA)

The ***suffix array*** $\mathsf{SA}[1..n]$ of $w[1..n]$ is an integer array, where each entry $\mathsf{SA}[i]$ points to the starting position of the i-th lexicographically least suffix of w. An example of

[1]Note that the term period, as used in the definition of a gapped repeat, differs from its earlier usage in the context of periodic strings. Throughout this thesis, the term period is consistently used in reference to periodic strings or runs.

a suffix array for the string $w[1..11] = mississippi$ is shown in Table 2.1.

Index(i)	**Sorted Suffixes**	SA	LCP
1	i	11	0
2	$ippi$	8	1
3	$issippi$	5	1
4	$ississippi$	2	4
5	$mississippi$	1	0
6	pi	10	0
7	ppi	9	1
8	$sippi$	7	0
9	$sissippi$	4	2
10	$ssippi$	6	1
11	$ssissippi$	3	3

Table 2.1: SA and LCP for the string $w[1..11] = mississippi$, the suffixes of string w are lexicographically sorted.

2.2.2 Longest Common Prefix Array (LCP)

The ***longest common prefix array*** $\mathsf{LCP}[1..n]$ of $w[1..n]$ is an array of integers that represent the length of the longest common prefix of consecutive suffixes in the SA; that is, between suffixes starting at $\mathsf{SA}[i]$ and $\mathsf{SA}[i-1]$ for all $2 \leq i \leq n$, assuming $\mathsf{LCP}[1] = 0$. See Table 2.1 for an example computed on $w[1..11] = mississippi$.

2.2.3 Border Array (β)

For a string $w[1..n]$, the array $\beta[1..n]$ is defined as ***border array*** of w if $\beta[i]$ represents the length of the longest border of the prefix $w[1..i]$ for $1 \leq i \leq n$. Note that $\beta[1] = 0$ since $w[1]$ has an empty border ε. An example of a string $w[1..11] = abracadabra$ and its corresponding border array β is shown in Table 2.2.

Index(i)	1	2	3	4	5	6	7	8	9	10	11
$w[i]$	a	b	r	a	c	a	d	a	b	r	a
$\beta[i]$	0	0	0	1	0	1	0	1	2	3	4

Table 2.2: The border array β of string $w[1..11] = abracadabra$.

2.2.4 Open-Closed Sequence (OC)

The open-closed sequence (or OC-sequence) of a word was first formally defined by De Luca [21] in 2017. Given a string $w[1..n]$, the ***open-closed sequence*** $\mathsf{OC}[1..n]$ is a binary array where each entry is defined as:

$$\mathsf{OC}[i] = \begin{cases} 1, & \text{if the prefix } w[1 \dots i] \text{ is closed,} \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

where $1 \leq i \leq n$. The open-closed sequence OC for the string $w[1..11] = abracadabra$ is illustrated in Table 2.3.

Index(i)	1	2	3	4	5	6	7	8	9	10	11
$w[i]$	a	b	r	a	c	a	d	a	b	r	a
$\mathsf{OC}[i]$	1	0	0	1	0	0	0	0	1	1	1

Table 2.3: The open-closed sequence OC of string $w[1..11] = abracadabra$.

2.3 Suffix Tree

The ***suffix tree*** $\mathcal{T}_w$ for a string $w[1..n]$ is a rooted, directed acyclic compressed trie constructed over the string w. The tree contains exactly n leaves, each uniquely labeled from 1 to n, corresponding to the suffixes $w[i..n]$ for $1 \leq i \leq n$. Each internal node has at least two children. Every edge in the tree is labeled with a non-empty substring of w, and no two edges originating from the same node begin with the same character. The key property of the suffix tree is that for any leaf node labeled i, the concatenation of the edge labels on the path from the root to i, exactly spells out the suffix $w[i..n]$. Figure 2.1 illustrates a suffix tree constructed for a string $w[1..12] = mississippi\$$.

To ensure uniqueness and proper termination of all suffixes, w is assumed to end with a special ***sentinel*** character \$, which does not occur elsewhere in the string and is assumed to be lexicographically least in alphabet Σ. The sentinel character \$ ensures that no suffix is a prefix of another, allowing the tree to have exactly one unique path for each suffix and ensuring that all suffixes terminate at leaves.

Observe that the suffix array SA can be obtained directly by performing a left-to-right depth-first traversal of the suffix tree and recording the leaf labels in the order in which they are encountered. These leaf node labels correspond to the starting positions of the lexicographically sorted suffixes of w.

2.4 The Extended Dis-joint Set

In 2015, Kociumaka et al. [31] introduced an extension to the classic disjoint-set data structure. Before describing this extended structure, let's briefly recall the classical disjoint-set data structure, also known as the union-find data structure. The classic disjoint-set data structure maintains a collection of non-overlapping sets, where each set is identified by a distinguished member called its ***representative***. The operation *Make-Set*(x) creates a new set containing only the element x, assuming that x does

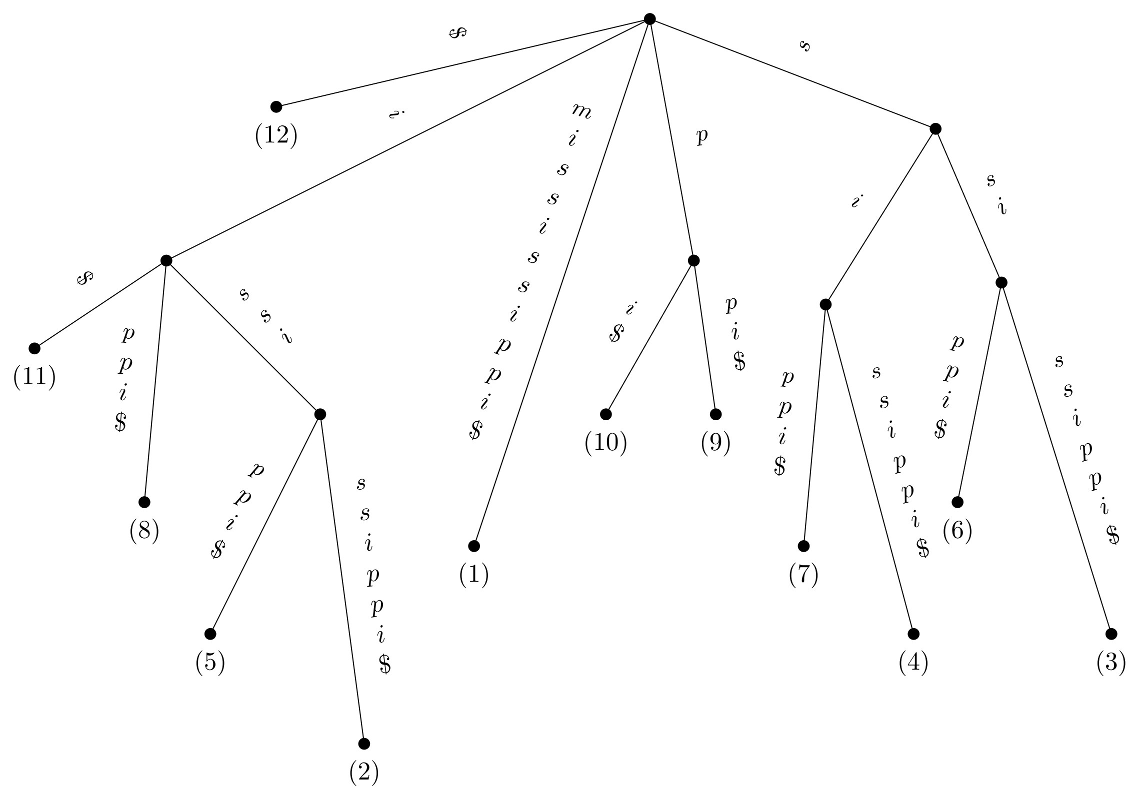

Figure 2.1: An example of a suffix tree ($\mathcal{T}_w$) for a string $w[1..12] = mississippi\$$.

not already belong to another set. *Find*(x) returns the representative of the set containing x. The operation *Union*(x, y) merges the sets containing x and y into a single set, provided they are disjoint. The resulting set may adopt either representative, and conceptually replaces the two original sets in the collection. This structure is fundamental in algorithms that manage partitioned data, such as graph connectivity and Kruskal's minimum spanning tree algorithm. For further details, see Chapter 21 of Introduction to Algorithms by Cormen, Leiserson, Rivest, and Stein (CLRS) [17].

Now we describe the extended disjoint-set data structure. For a set X of integers and $x \in X$, the function *next* is defined as:

$$next_X(x) = \min\{y \in X \mid y > x\}, \tag{2.2}$$

and assuming $next_X(x) = \infty$ if $x = \max(X)$. Given a partition $\mathcal{P} = \{P_1, \ldots, P_k\}$ of $U = \{1, \ldots, n\}$ and $\mathcal{P}' = \{\bigcup \mathcal{P}\}$ representing the partition after the union operation, the ***ChangeList*** is defined as:

$$ChangeList(\mathcal{P}, \mathcal{P}') = \{(x, next_{\mathcal{P}'}(x)) : next_{\mathcal{P}}(x) \neq next_{\mathcal{P}'}(x)\}. \tag{2.3}$$

where, $next_{\mathcal{P}}(x) = next_{P_i}(x)$ where $x \in P_i$. Note that pairs are not included in the *ChangeList* when one of its values is ∞.

The *Find* operation simply returns the corresponding set representative. The *Union* operation merges sets to compute the updated partition $\mathcal{P}'$ and also computes the associated *ChangeList*.

For example, given $\mathcal{P} = \{\{1, 2, 4\}, \{3, 5\}\}$ and after a *Union* operation resulting in $\mathcal{P}' = \{\{1, 2, 3, 4, 5\}\}$:

$next_{\mathcal{P}}(1) = 2, next_{\mathcal{P}'}(1) = 2$

$next_{\mathcal{P}}(2) = 4, next_{\mathcal{P}'}(2) = 3$

$next_{\mathcal{P}}(3) = 5, next_{\mathcal{P}'}(3) = 4$

$next_{\mathcal{P}}(4) = \infty, next_{\mathcal{P}'}(4) = 5$

$next_{\mathcal{P}}(5) = \infty, next_{\mathcal{P}'}(5) = \infty$

$ChangeList(\mathcal{P}, \mathcal{P}') = \{(2, 3), (3, 4), (4, 5)\}$

The pseudo-code for the *Union* operation is described in [31]. A detailed illustration of the extended disjoint-set data structure - in the context of this thesis - is provided in Section 4.4 and implementation details are provided in Section 6.2

Chapter 3

Related Work

3.1 Closed Strings

In 2011, Fici introduced the concept of open and closed strings in [22], where he explored the relationship between Trapezoidal and Sturmian words. He proved that open Trapezoidal words are necessarily primitive, while closed Trapezoidal words correspond to Sturmian words. Additionally, he established that Trapezoidal palindromes are closed and thus Sturmian. The notion of open and closed strings provides a framework for characterizing strings based on their combinatorial and structural properties. In [15], Bucci et al. (2013) studied the prefixes of the Fibonacci word in relation to their classification as open or closed Trapezoidal words. It is shown that the sequence of open and closed prefixes of the Fibonacci word corresponds to the Fibonacci sequence.

In [6], Badkobeh et al. (2014) explore algorithmic challenges related to closed factors in strings. One of the problems they address is the *Longest Closed Factorization* problem, which involves greedily decomposing a string into a sequence of its longest closed factors. They present an algorithm that solves this problem in $\mathcal{O}(n)$ time and space, where n is the length of the string. Another problem discussed is the *Longest Closed Factor Array (LCF)* problem, which entails computing the length of the longest closed factor starting at each position in the string. Using computational geometry techniques, the authors achieve a solution with a time complexity of $\mathcal{O}(n \log n / \log \log n)$. Later, in 2015, Bannai et al. [11], demonstrated that reconstructing a string from its LCF array is simpler than constructing or verifying the array. Furthermore, the reconstructed string is unique. They also improved the time complexity for construction and verification, reducing it to $\mathcal{O}(n\sqrt{\log n})$.

In [7], Badkobeh et al. (2015) prove that a string of length n has at least $n+1$ distinct closed factors and provide a characterization of the strings that have exactly $n+1$ closed factors. Additionally, they show that the number of distinct closed factors in a string of length n can be as large as $\mathcal{O}(n^2)$. In [8], Badkobeh et al. (2016) introduce several linear-time algorithms (in addition to the ones introduced in [6]), including factorizing a string into a sequence of shortest closed factors of length at least two, computing the shortest closed factor of length at least two starting from each position in the string, and determining a minimal closed factor of length at least two that includes every position of the string.

The *oc-sequence* of a string is defined as a binary sequence where the i-th element is 0 if the prefix of length i of the string is open, and 1 if it is closed. In [23], Fici (2017) proposed a linear-time algorithm to compute the oc-sequence of any finite string and in [21] De Luca et al. (2017) present a linear-time algorithm to reconstruct a finite Sturmian string from its oc-sequence. This sequence is closely connected to the combinatorial and periodic structure of a string, providing valuable insights into its properties.

In [1], Alamro et al. (2017) introduce the k-closed string problem, which generalizes the classical closed string problem by allowing approximate matches. Two strings of equal length are said to have a Hamming distance of k, if they differ in exactly k positions. Using this notion, a k-closed string permits up to k mismatches between its prefix and suffix borders. The authors present an algorithm with time complexity $\mathcal{O}(kn)$ and space complexity $\mathcal{O}(n)$, supported by implementation details and experimental evaluation.

Alzamel et al. (2019) [2] present an on-line algorithm for computing the Longest Closed Factorization with an amortized time complexity of $\mathcal{O}(\log n)$ per character, where n is the length of the string. They also introduce the Minimum Closed Factorization problem, which seeks the smallest number of closed factors that cover w. For this problem, they describe an off-line algorithm with a time complexity of $\mathcal{O}(n \log^2 n)$, as well as an on-line algorithm.

Arnoux-Rauzy words were first introduced in [5] in the context of a 3-letter alphabet. They serve as a natural generalization of Sturmian words to alphabets with more than two letters. Parshina and Zamboni in [41] (2019) focus on counting the number of closed factors of each length in an infinite Arnoux-Rauzy word. An explicit formula is derived for this count, leading to the result that the number of closed factors of length n grows without bound as n approaches infinity.

In [28], Jahannia et al. (2022) introduce the concept of closed z-factorization, building on Ziv–Lempel factorization, for both finite and infinite words. They determine the closed z-factorization of infinite m-bonacci words for all $m \geq 2$ and provide a classification of the closed prefixes of these infinite m-bonacci words.

In [9], Badkobeh et al. (2022) introduce the concept of a maximal closed substring (MCS). MCSs with an exponent of at least 2 are typically referred to as runs, while those with an exponent less than 2 are special cases of maximal gapped repeats. The authors use the oc-sequence to show that a string of length n contains at most $\mathcal{O}(n\sqrt{n})$ MCSs. They also provide an output-sensitive algorithm to identify all m MCSs in a string of length n over a constant-size alphabet, which runs in $\mathcal{O}(n \log n + m)$ time using suffix trees. Later in [10], Badkobeh et al. (2024) extend their earlier algorithm to handle strings over a general alphabet with a time complexity of $\mathcal{O}(n \log n)$. This implicitly improves the bound on the total number of MCSs. Kosolobov [34] (2024) later proved this bound directly.

In [40], Parshina et al. (2024) show that the maximal number of distinct closed factors in a word of length n is $\frac{n^2}{6}$. They define infinite closed-rich words as those where every factor of length n contains a quadratic number of distinct closed factors. The authors provide necessary and sufficient conditions for a word to be closed-rich, showing that all linearly recurrent words are closed-rich and characterizing rich words among Sturmian words.

In [37], Mieno et al. (2025) present a linear algorithm to count the number of distinct closed factors of a string in $\mathcal{O}(n \log \sigma)$ time and another in $\mathcal{O}(n)$ time. They also describe methods to enumerate all distinct closed factors, achieving a complexity of $\mathcal{O}n^2)$, which is further optimized to $\mathcal{O}(n\sqrt{\log n} + \text{output} \cdot \log n)$ using weighted ancestor queries (WAQ).

3.2 Covers

In their comprehensive survey, Mhaskar and Smyth (2022) synthesize decades of work on string covers, short substrings that collectively cover a larger string by their adjacent and/or overlapping occurrences [35]. They trace the origin of this concept back to Apostolico and Ehrenfeucht (1993), who first coined the term quasiperiodicity and developed efficient detection methods [3]. A string is superprimitive if it is not quasiperiodic, Apostolico, Farach, and Iliopoulos (1991) tackled problems of superprimitivity, laying the groundwork for optimal cover-finding algorithms [4]. Breslauer's 1992 online algorithm further improved these methods, leading to a suite of linear- and near-linear-time solutions—including the work of Iliopoulos, Moore, and Park (1996) that computed all string covers in $\mathcal{O}(n \log n)$ time [26]. Mhaskar and Smyth (2022) in their survey, not only organized these milestones but also charted future directions in this rapidly evolving area of combinatorial string processing.

In this thesis, the primary focus on the covers refers to their context within Fibonacci words, as defined and discussed in Chapter 5. The following papers have made significant contributions to the study of covers in Fibonacci words:

Iliopoulos et al. [27] (1998) studied the covers of circular Fibonacci words, a class of sequences relevant to combinatorial pattern matching. They also introduced a linear-time algorithm to efficiently enumerate these covers, marking a notable contribution to the computational analysis of circular patterns.

Christou et al. [16] (2016) examined the structural properties of Fibonacci words and identified all their covers. Their study provided a better understanding of periodicity in Fibonacci words and described their quasiperiodic behavior, contributing to a basis for further research in the area.

Crochemore et al. [19] (2021) studied cyclic shifts of strings and developed efficient algorithms to determine the shortest covers of all such shifts. Their findings showed that the number of distinct cover lengths for Fibonacci words increases linearly, providing valuable insights into the combinatorial properties of these sequences.

More recently, Radoszewski and Zuba [42] (2024) introduced a sublinear-time algorithm to compute all covers of a string over an integer alphabet. They also provided an in-depth analysis of cover arrays specific to Fibonacci words. Leveraging the repetitive structure of Fibonacci words, they identified a precise pattern describing how the covers of each prefix evolve within the Fibonacci sequence.

3.3 Gapped-Repeats

The study of gapped repeats began with two foundational works. Brodal et al. [13] (1999) addressed the problem of finding all maximal pairs with bounded gap in a string. Given a string of length n, their algorithm reports all such pairs in time $\mathcal{O}(n \log n + z)$, where z is the number of output pairs. The method relies on suffix tree constructions combined with auxiliary balanced data structures to identify all right-maximal pairs satisfying the given gap constraints.

In parallel, Kolpakov and Kucherov [33] (2000) studied the problem of repeats with a fixed gap, where the goal is to find all factors of the form $u\,v\,u$ such that the gap v has a specified fixed length d. They proposed an algorithm with a time complexity of $\mathcal{O}(n \log d + S)$, where S is the size of the output. Their algorithm efficiently enumerates all fixed-gap repeats and builds on standard string indexing techniques to identify such patterns precisely.

Together, these two papers provide the algorithmic foundation for the study of repeats under fixed or bounded-gap constraints. They form the basis for much of the later work in gapped pattern matching in combinatorics on words and theoretical computer science.

Following the foundational work on fixed-gap and bounded-gap repeats, several studies have expanded and refined the problem space [18, 24, 44]. These investigations introduced new algorithms and structural insights that deepened our understanding of gapped patterns in strings. Recently, Yamane et al. [48] (2025) investigated maximal gapped repeats in Fibonacci strings. They identified a structural characterization of the arms of such repeats, showing that they fall into two classes: Fibonacci arms and palindromic arms. Using these forms, the authors derived an upper bound on the number of maximal gapped repeats in the k-th Fibonacci word. Their work also established that Fibonacci words form a new class of strings exhibiting a rich structure of maximal gapped repeats. These results form a foundation for our study in Chapter 5, where we explore the intersection of these patterns with closed string properties and introduce the notion of gapped-MCSs in Fibonacci words.

Chapter 4

Closed Substrings

4.1 Naive Computation of All Closed Substrings

All closed substrings of a string $w[1..n]$ can be computed naively by calculating the OC_w sequence for each suffix of w. First, the border array β_w of any given string can be computed in $\mathcal{O}(n)$ time [43]. Once the border array is known, the open-closed sequence OC_w can be computed in $\mathcal{O}(n)$ time [23]. Thus, the overall time complexity to compute all closed substrings naively is $\mathcal{O}(n^2)$. Table 4.1 provides an illustrative example of this computation. Given the border array β_w of a string $w[1..n]$, the open-closed sequence $\mathsf{OC}_w[i]$ can be computed according to the following equation:

$$\mathsf{OC}_w[i] = \begin{cases} 1, & \text{if } i = 1 \text{ or } \beta_w[i] \text{ is maximum and unique among } \{\beta_w[j] \mid 1 \leq j \leq i\}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

Index (i)	1	2	3	4	5	6	7	8	9	10	11	Closed Subtrings
$w[1..11]$	m	i	s	s	i	s	s	i	p	p	i	
$\mathsf{OC}_{w[1..11]}$	1	0	0	0	0	0	0	0	0	0	0	m
$\mathsf{OC}_{w[2..11]}$		1	0	0	1	1	1	1	0	0	0	i, $issi$, $issis$, $ississ$, $ississi$
$\mathsf{OC}_{w[3..11]}$			1	1	0	0	1	1	0	0	0	s, ss, $ssiss$, $ssissi$
$\mathsf{OC}_{w[4..11]}$				1	0	1	0	1	0	0	0	s, sis, $sissi$
$\mathsf{OC}_{w[5..11]}$					1	0	0	1	0	0	0	i, $issi$
$\mathsf{OC}_{w[6..11]}$						1	1	0	0	0	0	s, ss
$\mathsf{OC}_{w[7..11]}$							1	0	0	0	0	s
$\mathsf{OC}_{w[8..11]}$								1	0	0	1	i, $ippi$
$\mathsf{OC}_{w[9..11]}$									1	1	0	p, pp
$\mathsf{OC}_{w[10..11]}$										1	0	p
$\mathsf{OC}_{w[11..11]}$											1	i

Table 4.1: Naive computation of all closed substrings of $w[1..11] = mississippi$.

4.2 Compact Representation of Closed Substrings

A string $w[1..n]$ contains $\mathcal{O}(n^2)$ closed substrings, which can naively be computed in $\mathcal{O}(n^2)$ time by computing the border of all substrings of w. In this section, we define a compact representation for all $\mathcal{O}(n^2)$ closed substrings in Theorem 1 that requires only $\mathcal{O}(n \log n)$ space. Algorithm 2 in Section 4.6 presents an $\mathcal{O}(n \log n)$ time solution to compute all closed substrings of w in a compact representation of size $\mathcal{O}(n \log n)$.

Let u and v be the proper closed prefixes of the maximal right-closed substring r. Then, by definition of right-extendibility we define an equivalence relation R as follows:

$$R = \{(u, v) \mid u \text{ and } v \text{ are both right-extendible to } r\}.$$

Next, we define the equivalence class E_r of r under the relation R as follows:

$$E_r = \{u \mid u \text{ is right-extendible to } r\}. \tag{4.2}$$

Lemma 1. *Given a string $w[1..n]$, if $w[1..m]$ is the shortest maximal right-closed prefix of w, then $w[1..m] = \lambda^m$, where $\lambda \in \Sigma$ and $1 \leq m \leq n$. Moreover, each prefix of λ^m is also closed.*

Proof. We prove the lemma in two parts. First part by contradiction. Suppose that $w[1..m]$ is the shortest maximal right-closed prefix of w, and that $w[1..m] \neq \lambda^m$. W.l.o.g let $w[1] = \lambda$. Let $1 < j \leq m$ be the smallest index such that $w[j] = \lambda_1$ and $w[1..j-1] = \lambda^{j-1}$, where $\lambda_1 \neq \lambda \in \Sigma$. Then, $w[1..j] = \lambda^j \lambda_1$. However, by definition of maximal right-closed string, $w[1..j-1] = \lambda^{j-1}$ is a shorter maximal right-closed string of w — a contradiction.

Second, for any string of the form $u = \lambda^m$, consider its prefixes $p = u[1..j]$, where $1 \leq j \leq m$. If $j = 1$, then p is closed trivially. If $j \geq 2$, then the border length of each prefix p is equal to $j - 1$, which is maximum and unique. Thus, all the prefixes of $u = \lambda^m$ are closed. $\square$

Let r denote a maximal right-closed substring, b its border, and p its shortest closed prefix such that $p = r$ or $p \in E_r$. Let $r_1, r_2, \ldots, r_{K_i}$ be all the maximal right-closed substrings starting at index i of a string $w[1..n]$, where $1 \leq i \leq n$, and K_i is the number of maximal right-closed substrings starting at index i, such that $|r_1| < |r_2| < \cdots < |r_{K_i}|$.

In Equation (4.3) we present a formula to compute the length of the shortest closed prefix p_j of r_j such that $p_j = r_j$ or $p_j \in E_{r_j}$, where $1 \leq j \leq K_i$. Lemma 2

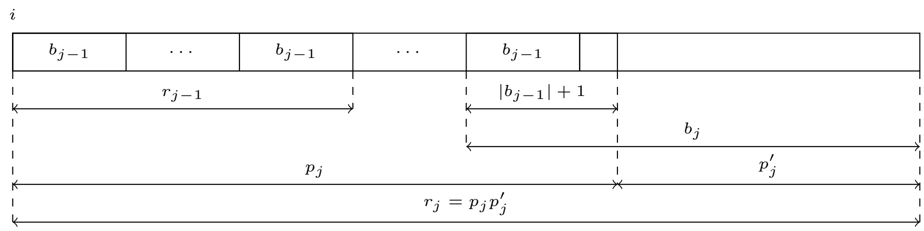

Figure 4.1: Illustration of the second case in the proof of Lemma 2, showing that $|p_j|$ can be computed from $|b_j|$, $|b_{j-1}|$ and $|r_j|$, such that $p_j = r_j$ or $p_j \in E_{r_j}$. Note that when $p_j = r_j$, $p'_j = \varepsilon$ [29].

proves Equation (4.3) and is illustrated in Figure 4.1.

Lemma 2. *Let $1 \leq j \leq K_i$. The length of the shortest closed prefix p_j of r_j, such that $p_j = r_j$ or $p_j \in E_{r_j}$, is given by:*

$$|p_j| = \begin{cases} 1, & \textit{if } j = 1, \\ |r_j| - |b_j| + |b_{j-1}| + 1, & \textit{if } 1 < j \leq K_i. \end{cases} \tag{4.3}$$

where b_j and b_{j-1} are the longest borders of r_j and r_{j-1}, respectively.

Proof. We prove the Lemma by considering two cases. For $j = 1$, by Equation (4.2) and Lemma 1, all prefixes of r_1 are closed and $|p_1| = 1$.

For $1 < j \leq K_i$, by definition of a closed substring, the shortest closed prefix p_j starting at index i and longer than r_{j-1} (whose longest border has length $|b_{j-1}|$) must have the longest border of length exactly $|b_{j-1}| + 1$. Either $p_j = r_j$ or by definition of right-extendibility p_j extends to $r_j = p_j p'_j$, with the longest border b_j, and each prefix of r_j with length at least $|p_j|$ is closed. Since $|p'_j| = |b_j| - (|b_{j-1}| + 1)$, we have $|r_j| = |p_j| + |p'_j| = |p_j| + |b_j| - (|b_{j-1}| + 1)$. Solving for $|p_j|$, we obtain the desired formula: $|p_j| = |r_j| - |b_j| + |b_{j-1}| + 1$. $\square$

Lemma 3 (Theorem 1 in [34]). *A string $w[1..n]$ contains at most $\mathcal{O}(n \log n)$ maximal right (left)-closed substrings.*

Theorem 1 follows directly from Equation (4.2), Lemma 2 and 3.

Theorem 1. *The compact representation of all closed substrings of $w[1..n]$ requires at most $\mathcal{O}(n \log n)$ space and is given by:*

$$\mathcal{C}(w) = \{(i, |p_j|, |r_j|) \mid 1 \leq i \leq n,\ 1 \leq j \leq K_i\}, \tag{4.4}$$

where K_i is the number of maximal right-closed substrings starting at position i, and for each j-th such substring r_j, starting at index i, p_j is the shortest closed prefix of r_j such that $p_j = r_j$ or $p_j \in E_{r_j}$.

The set of all closed substrings of $w[1..n]$, denoted by Closed(w), can be enumerated using the following equation:

$$\text{Closed}(w) = \bigcup_{(i,|p_j|,|r_j|) \in \mathcal{C}(w)} \{w[i \ldots i + \ell - 1] \mid \ell \in [|p_j|, |r_j|]\}. \tag{4.5}$$

4.3 Maximal Right-Closed $\mathcal{MRC}$ Array

In this section, we introduce the primary data structure –the $\mathcal{MRC}$ array– which is used to compute all closed substrings and MCSs in $\mathcal{O}(n \log n)$ time.

Definition 1 ($\mathcal{MRC}$ Array[1]). *Given a string $w[1..n]$, the **maximal right-closed array ($\mathcal{MRC}$)** is an array of size n, where $\mathcal{MRC}[i]$, $1 \leq i \leq n$, stores the list of ordered pairs $(|r_j|, |b_j|)$, where $j = K_i$ to 1.*

[1]This definition uses the notation introduced in Section 4.2.

$C(w)$	**All Closed Factors**	**Maximal Right Closed Factors**	**MCS?**
$(1,1,1)$	m	m	Yes
$(2,4,7)$	$issi$, $issis$, $ississ$, $ississi$	$ississi$	Yes
$(2,1,1)$	i	i	Yes
$(3,5,6)$	$ssiss$, $ssissi$	$ssissi$	No
$(3,1,2)$	s, ss	ss	Yes
$(4,5,5)$	$sissi$	$sissi$	No
$(4,3,3)$	sis	sis	Yes
$(4,1,1)$	s	s	No
$(5,4,4)$	$issi$	$issi$	No
$(5,1,1)$	i	i	Yes
$(6,1,2)$	s, ss	ss	Yes
$(7,1,1)$	s	s	No
$(8,4,4)$	$ippi$	$ippi$	Yes
$(8,1,1)$	i	i	Yes
$(9,1,2)$	p, pp	pp	Yes
$(10,1,1)$	p	p	No
$(11,1,1)$	i	i	Yes

Table 4.2: Compact representation $\mathcal{C}(w)$ of $w[1..11] = mississippi$ with corresponding closed factors, maximal right-closed factors, and MCSs [29].

We can analogously define the ***maximal left-closed array (*** $\mathcal{MLC}$ ***)*** for maximal left-closed substrings ending at each index position of string $w[1..n]$. Table 4.3 shows an example of the $\mathcal{MRC}$ array of the string $w[1..11] = mississippi$.

4.4 $\mathcal{MRC}$ Array using Suffix Trees

To compute the $\mathcal{MRC}$ array of a string $w[1..n]$ using its suffix tree $\mathcal{T}_w$ (except the root node), we perform a bottom-up traversal of all internal nodes in $\mathcal{T}_w$. The idea

Index (i)	$\mathcal{MRC}[i]$
1	$(1, 0)$
2	$(7, 4), (1, 0)$
3	$(6, 3), (2, 1)$
4	$(5, 2), (3, 1), (1, 0)$
5	$(4, 1), (1, 0)$
6	$(2, 1)$
7	$(1, 0)$
8	$(4, 1), (1, 0)$
9	$(2, 1)$
10	$(1, 0)$
11	$(1, 0)$

Table 4.3: The $\mathcal{MRC}$ array for the string $w[1..11] = mississippi$ [29].

is to detect all maximal right-closed substrings of w and organize them, along with their border lengths, into the $\mathcal{MRC}$ array.

Let q be an internal node of $\mathcal{T}_w$, excluding the root, of a string $w[1..n]$. Let $u = \textit{path-label}(q)$ be the concatenation of edge labels from the root to q. The ***string-depth*** of q is defined as $\textit{string-depth}(q) = |u|$. The set of leaf nodes in the subtree rooted at q corresponds to the starting index positions of all occurrences of the substring u in w. Consider three such leaf nodes ln_1, ln_2, and ln_3 under q, where ln_1 and ln_2 are reachable via the same outgoing edge from q, and ln_3 is reachable via a different edge. Then, the occurrences of u corresponding to ln_1 and ln_2 are followed by the same character in w, whereas the occurrence corresponding to ln_3 is followed by a different character.

We begin by initializing n AVL trees, one for each leaf node of $\mathcal{T}_w$. Each AVL tree stores the position corresponding to its leaf, and all leaf nodes are initially marked as *processed*. Internal nodes are processed in a bottom-up order: a node q is processed only when all its descendant nodes have already been marked as processed. This ensures that we encounter longer substrings (those near the leaves) before shorter ones (higher up in the tree), naturally ordering maximal right-closed substrings by descending length in the final $\mathcal{MRC}$ array.

Let u denote the path-label from the root to q. The set of leaf nodes in the subtree rooted at q gives all starting positions of the substring u in w. To identify closed substrings, we sort these starting positions in ascending order. Any pair of consecutive occurrences of u forms a closed string.

However, to ensure that the identified closed strings are *maximal right-closed*, we further restrict our attention to consecutive occurrences of u that differ in the character immediately following u in w. This distinction is determined by whether the leaf nodes corresponding to those occurrences are reachable via different outgoing edges from node q in the suffix tree. If so, then the characters to the right of u at those positions are guaranteed to differ, forming the resulting maximal right-closed substring.

At each internal node q, we compute the *Union* of all AVL trees associated with its children. This merged structure is assigned to the internal node q. The *Union* operation also computes the *changeList* for node q, which identifies valid pairs of consecutive occurrences with different right extensions. This is facilitated by the extended-disjoint data structure (see Section 2.4 for more details), which maintains a next array throughout the *Union* operations, ensuring that the correct consecutive pairs are selected. Here, $\mathsf{next}[x] = next_{\mathcal{P}}(x)$ provides the next element of x within its set, enabling efficient identification of such consecutive pairs.

The $ChangeList$ at each node q then directly contributes entries to the $\mathcal{MRC}$ array. Let (x, y) be a pair in the $changeList$, representing two consecutive occurrences of the substring $u = \textit{path-label}(q)$ in w with different right extensions. The corresponding maximal right-closed substring starts at index x and has length $y + |u| - x$, while the length of its longest border is equal to the string-depth of q, i.e., $|u|$. As a result of the bottom-up traversal, longer maximal right-closed substrings are processed before shorter ones, ensuring that entries in each $\mathcal{MRC}[i]$ are stored in descending order of length.

Figure 4.2 illustrates the suffix tree $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$. The internal nodes are processed in bottom-up order from node A to node F and each node is annotated with its resulting $ChangeList$. Table 4.4 lists the path-label and string-depth associated with each internal node(except the root node), which are used to determine the border length of each such maximal right-closed substring.

Node(q)	$u = \textit{path-label}(q)$	$\textit{string-depth}(q)$
A	`issi`	4
B	`i`	1
C	`p`	1
D	`si`	2
E	`ssi`	3
F	`s`	1

Table 4.4: Path-labels and string-depths of all internal nodes of $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$.

At each internal node, the $\mathsf{next}[i]$ array is updated via *Union* operations over the AVL trees of its child nodes, as shown in Table 4.5. The *Union* operation also computes the $ChangeList$ (Table 4.6) to identify valid consecutive occurrences of

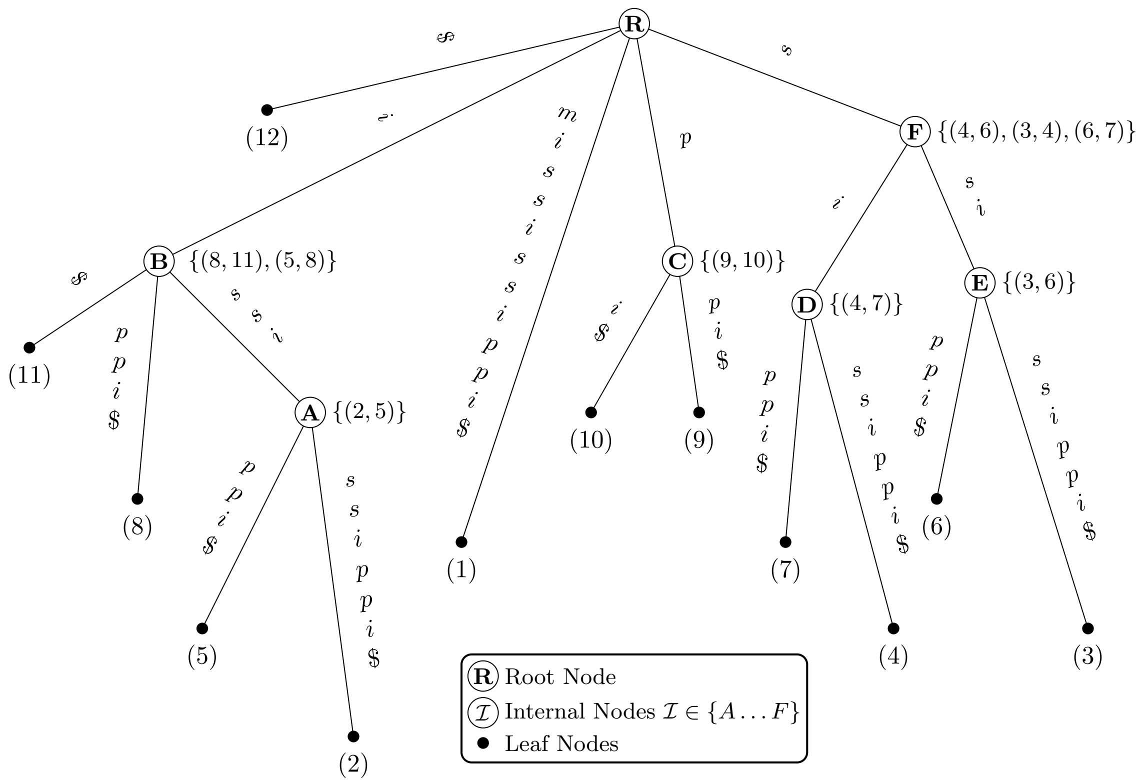

Figure 4.2: Suffix tree $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$. Internal nodes are processed in bottom-up order from A to F, each internal node is annotated with its computed *ChangeList*, which identifies the maximal right-closed substrings.

substrings with differing right extensions; thus identifying maximal right-closed substrings.

So far, we have computed maximal right-closed substrings of length at least 2 using the suffix tree. To complete the construction of the $\mathcal{MRC}$ array, we perform a simple linear scan of the string to identify single-character substrings that are also maximal right-closed. A character qualifies if it is followed by a different character or appears at the end of the string. These singletons are then added directly to $\mathcal{MRC}$.

The suffix tree for a string of length n contains exactly n leaf nodes and at most $n - 1$ internal nodes, as first established in Weiner's foundational work [46]. During

Node(q)	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
Initial	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
A	-1	5	-1	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	5	-1	-1	8	-1	-1	11	-1	-1	-1
C	-1	5	-1	-1	8	-1	-1	11	10	-1	-1
D	-1	5	-1	7	8	-1	-1	11	10	-1	-1
E	-1	5	6	7	8	-1	-1	11	10	-1	-1
F	-1	5	4	6	8	7	-1	11	10	-1	-1

Table 4.5: Tracing of the $\mathsf{next}[i]$ array for each internal node of $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$ during the computation of its $\mathcal{MRC}$ array.

Node(q)	$\mathcal{P}$	$\mathcal{P}'$	$ChangeList(\mathcal{P}, \mathcal{P}')$
A	$\{\{2\}, \{5\}\}$	$\{\{2, 5\}\}$	(2, 5)
B	$\{\{2, 5\}, \{8\}, \{11\}\}$	$\{\{2, 5, 8, 11\}\}$	(8, 11), (5, 8)
C	$\{\{9\}, \{10\}\}$	$\{\{9, 10\}\}$	(9, 10)
D	$\{\{4\}, \{7\}\}$	$\{\{4, 7\}\}$	(4, 7)
E	$\{\{3\}, \{6\}\}$	$\{\{3, 6\}\}$	(3, 6)
F	$\{\{3, 6\}, \{4, 7\}\}$	$\{\{3, 4, 6, 7\}\}$	(4, 6), (3, 4), (6, 7)

Table 4.6: *ChangeList* computation at each internal node of $\mathcal{T}_w$ for the string $w[1..12] = mississippi\$$.

the bottom-up traversal, we perform a sequence of *Union* operations over AVL trees (or any other balanced search structure) attached to the children of each internal node. The *Union* operation correctly computes the ChangeList, and any sequence of *Union* operations takes $\mathcal{O}(n \log n)$ time in total [31]. Furthermore, the single-characters that are maximal right-closed are computed by a linear scan on w, taking $\mathcal{O}(n)$ time. Thus, the overall time complexity for constructing the $\mathcal{MRC}$ array is $O(n \log n)$.

A related approach using AVL trees to compute maximal closed substrings is presented in [9, 10]; however, they do not have a formal algorithm in place. Moreover, while suffix trees are powerful in nature, they are space-intensive in practice. Therefore, in the following section, we design a more space-efficient solution that uses SA and LCP arrays to construct the $\mathcal{MRC}$ array.

4.5 $\mathcal{MRC}$ Array using SA and LCP

Algorithm 1 computes the $\mathcal{MRC}$ array by first identifying all the maximal right-closed substrings of length greater than one. It uses a stack to store AVL trees created during its execution. The algorithm scans the LCP array from left to right. Beginning with $\mathsf{LCP}[1] = 0$, and for increasing $\mathsf{LCP}[i]$ values, the algorithm creates a single node (root) AVL tree with $key = \mathsf{SA}[i]$ and $value = \mathsf{LCP}[i]$ and pushes it onto the stack, until either the LCP value decreases or the end of the array is reached.

We denote by LCP_{max} the value stored in the root node of the AVL tree on the top of the stack. When a decrease in the LCP value (or the end of the array) is encountered, the algorithm pops a collection of AVL trees from the stack, denoted as $\mathcal{L}_{suffixSets}$, such that all suffixes represented by an AVL tree key share the same longest common prefix equal to LCP_{max}. Since LCP_{max} is the maximum LCP value of all the AVL trees on the stack, popping this collection, $\mathcal{L}_{suffixSets}$, preserves the strictly decreasing order of ordered pairs in the $\mathcal{MRC}$ array. This collection is then merged into a single AVL tree using the *Union* operation presented in [31][2], resulting in $\mathcal{T}_{suffixes} = \bigcup \mathcal{L}_{suffixSets}$. The *Union* operation also computes a list of ordered pairs representing maximal right-closed substrings via the $ChangeList(\mathcal{L}_{suffixSets}, \mathcal{T}_{suffixes})$ operation defined as follows:

[2]The *Union* operation, defined in [31], builds upon foundational methods from [12, 14].

$$ChangeList(\mathcal{P}, \mathcal{P}') = \{(x, next_{\mathcal{P}'}(x)) : next_{\mathcal{P}}(x) \neq next_{\mathcal{P}'}(x)\}, \tag{4.6}$$

where $\mathcal{P}$ is a partition of the set $\mathcal{P}'$, and $next_X(x) = \min\{y \in X \mid y > x\}$, $x \in X$ and $X \subseteq \mathbb{Z}^+$. In particular, if $x \in P_i$ for some $P_i \in \mathcal{P}$, then $next_{\mathcal{P}}(x) = next_{P_i}(x)$. Each pair $(x, y) \in ChangeList(\mathcal{L}_{suffixSets}, \mathcal{T}_{suffixes})$ identifies a maximal right-closed substring $r_j = w[x..y+LCP_{max}-1]$, where LCP_{max} is the length of its longest border. Hence, the pair $(y + LCP_{max} - x, LCP_{max})$ is added to $\mathcal{MRC}[x]$. All such pairs from the *ChangeList* are added to the $\mathcal{MRC}$ array. The algorithm then assigns to the root node of the AVL tree, $\mathcal{T}_{suffixes}$, the LCP value of the AVL tree at the top of the stack and pushes it onto the stack, only if the stack is not empty–thus maintaining the lexicographic ordering of the set of suffixes–and continues scanning the LCP array.

To compute the maximal right-closed substrings of length one, it performs a simple linear scan of the string w and adds $w[i]$ to $\mathcal{MRC}[i]$, if $i = n \vee (1 \leq i < n \wedge w[i] \neq w[i+1])$.

It is known that LCP array identifies all the occurrences of the repeating substrings that cannot be extended to the right. Moreover, the suffix array groups all these occurrences within specific ranges. Algorithm 1 identifies such ranges by tracking the rise and fall of values in the LCP array, and stores the corresponding indices in $\mathcal{L}_{suffixSets}$ as keys in AVL trees, starting with the substring of length LCP_{max}. The *Union* operation then merges these single node AVL trees to form one AVL tree ($\mathcal{T}_{suffixes}$) that contains all the SA values; that is, all the starting positions of the substrings of length LCP_{max} in the string w.

The $ChangeList(\mathcal{L}_{suffixSets}, \mathcal{T}_{suffixes})$ operation then returns every pair of consecutive occurrences of substrings in the $\mathcal{L}_{suffixSets}$. These consecutive occurrences form

Algorithm 1 Compute $\mathcal{MRC}$ Array

Input: A string $w[1..n]$, its $\mathsf{SA}[1..n]$ and $\mathsf{LCP}[1..n]$.
Output: The $\mathcal{MRC}$ array of string $w[1..n]$.

$i \leftarrow 0$, $\mathcal{MRC} \leftarrow \{\emptyset\}^n$, $stack \leftarrow \emptyset$
while $stack \neq \emptyset$ **or** $i < n$ **do**
    while $i < n$ **and** ($stack = \emptyset$ **or** $\mathsf{LCP}[i] \geq LCP(top(stack))$) **do**
        $push(stack, AVLTree(\{\mathsf{SA}[i]\}, \mathsf{LCP}[i]))$
        $i \leftarrow i + 1$
    $\mathcal{L}_{suffixSets} \leftarrow \emptyset$
    $LCP_{max} \leftarrow LCP(top(stack))$
    while $stack \neq \emptyset$ **and** $LCP(top(stack)) = LCP_{max}$ **do**
        $insert(\mathcal{L}_{suffixSets}, pop(stack))$
    if $LCP_{max} \neq 0$ **then**
        $previousLCP \leftarrow LCP(top(stack))$
        $insert(\mathcal{L}_{suffixSets}, pop(stack))$
        $\mathcal{T}_{suffixes} = \bigcup \mathcal{L}_{suffixSets}$ ▷ $\bigcup \mathcal{L}_{suffixSets}$ returns an AVL tree
        foreach $(x, y) \in ChangeList(\mathcal{L}_{suffixSets}, \mathcal{T}_{suffixes})$ **do** ▷ Equation (4.6)
            $insert(\mathcal{MRC}[x], (y + LCP_{max} - x, LCP_{max}))$
        $LCP(\mathcal{T}_{suffixes}) \leftarrow previousLCP$
        $push(stack, \mathcal{T}_{suffixes})$
for $i \leftarrow 1$ **to** n **do**
    if $i = n$ **or** ($i < n$ **and** $w[i] \neq w[i+1]$) **then**
        $insert(\mathcal{MRC}[i], (1, 0))$
return $\mathcal{MRC}$

the border of the maximal right-closed substring and are therefore added to the $\mathcal{MRC}$ array. Finally, the simple linear scan adds maximal right-closed substrings of length one. The algorithm clearly computes the maximal right-closed substrings in strictly decreasing order of length as required in the $\mathcal{MRC}$ array. Thus, the correctness of Algorithm 1 follows.

We note that the algorithm in the worst case performs $4n - 2$ stack operations for a string $w = \lambda^n$ and in the best case performs $2n$ stack operations for a string with all distinct characters.

Lemma 4 ([31]). *The Union operation correctly computes the ChangeList, and any sequence of Union operations takes $\mathcal{O}(n \log n)$ time in total.*

Theorem 2. *Algorithm 1 correctly computes the $\mathcal{MRC}$ array of a string $w[1..n]$ using* SA *and* LCP *in $\mathcal{O}(n \log n)$ time.*

Proof. We prove the total time complexity of Algorithm 1 in three parts. First, for stack operations, we introduce a potential function $\Phi(S) = |S| \geq 0$, where $|S|$ is the number of elements in the stack. The actual cost of each push and pop operation is 1. Since we push all n single node AVL trees corresponding to each suffix onto the stack, the amortized cost for each push operation is $c'_{push} = 1 + \Delta\Phi(S) = 2$, resulting in a total amortized cost of $2n$. Additionally, in the algorithm, we pop at least two AVL trees and push their *Union* back onto the stack, and because the amortized cost of this sequence of pops and push in the worse case is $c'_{pops-push} = 3 + \Delta\Phi(S) = 2$. The maximum number of such operations is $n - 1$, leading to a total amortized cost of $2(n-1)$. Therefore, the total cost of all stack operations in the worst case is $4n-2$, which is $\mathcal{O}(n)$. Second, by Lemma 4, the time complexity of the sequence of *Union* operations is $\mathcal{O}(n \log n)$. Finally, the single-characters that are maximal right-closed are computed by a linear scan on w, taking $\mathcal{O}(n)$ time. Therefore, the total time complexity is $\mathcal{O}(n \log n)$. □

Reversing the string in Algorithm 1 yields the following corollary.

Corollary 1. *The $\mathcal{MLC}$ array of a string $w[1..n]$ can be computed correctly using* SA *and* LCP *in $\mathcal{O}(n \log n)$ time.*

4.6 Algorithm to Compute $\mathcal{C}(w)$

Algorithm 2 computes the compact representation $\mathcal{C}(w)$, as in Equation (2), of all closed substrings of a string $w[1..n]$, by simply traversing all n lists in the $\mathcal{MRC}$ array (see example in Table 4.2 in the Appendix). Since the $\mathcal{MRC}$ array is of size $\mathcal{O}(n \log n)$ we get Theorem 3.

Algorithm 2 Compute $\mathcal{C}(w)$

Input: $\mathcal{MRC}$ array of strings of $w[1..n]$

Output: Compact representation $\mathcal{C}(w)$ as in Equation (4.4)

```
C ← ∅
for i ← 1 to n do
    for j ← 1 to length of MRC[i] do
        (ℓr_j, ℓb_j) ← MRC[i][j]                ▷ ℓr_j = |r_j|, ℓb_j = |b_j|, ℓp_j = |p_j|
        if j = 1 then
            ℓp_j ← 1                            ▷ Equation (4.3)
        else
            (ℓr_{j−1}, ℓb_{j−1}) ← MRC[i][j − 1]
            ℓp_j ← ℓr_j − ℓb_j + ℓb_{j−1} + 1   ▷ Equation (4.3)
        insert(C, (i, ℓp_j, ℓr_j))
return C
```

Theorem 3. *Algorithm 2 correctly computes all $\mathcal{O}(n^2)$ closed substrings of a string $w[1..n]$ in a compact representation in $\mathcal{O}(n \log n)$ time.*

4.7 Computing All MCSs

Algorithm 3 computes all MCSs of a string $w[1..n]$ by iterating through the $\mathcal{MRC}$ array (see example in Table 4.2 in the Appendix). For each index i in w (where $1 \leq i \leq n$), the algorithm examines each maximal right-closed substring represented in $\mathcal{MRC}[i]$ and verifies whether the substring is also maximal left-closed using a constant time check $i = 1$ or ($i > 1$ and $w[i-1] \neq w[i + |r| - |b| - 1]$). If the condition is satisfied, the substring is added to the MCS list. Since scanning through the $\mathcal{MRC}$ array takes $\mathcal{O}(n \log n)$ time, and each check is performed in constant time, the algorithm computes all MCSs in $\mathcal{O}(n \log n)$ time, establishing Theorem 4.

Algorithm 3 Compute all MCSs

Input: A string $w[1..n]$ and $\mathcal{MRC}$ array

Output: All MCSs

```
1: mcsList ← ∅
2: for i ← 1 to n do
3:     for j ← 1 to length of MRC[i] do
4:         (ℓr, ℓb) ← MRC[i][j]                          ▷ ℓr = |r|, ℓb = |b|
5:         if i = 1 or (i > 1 and w[i − 1] ≠ w[i + ℓr − ℓb − 1]) then
6:             insert(mcsList, w[i..i + ℓr − 1])
7: return mcsList
```

Theorem 4. *Algorithm 3 correctly computes all MCSs of a string* $w[1..n]$ *in* $\mathcal{O}(n \log n)$ *time.*

Chapter 5

MCSs in Fibonacci Words

In this chapter, we study the occurrences of MCSs in a Fibonacci word (f_n), and provide a formula to compute the exact number of MCSs in f_n.

Let us recall that a ***Fibonacci*** word f_n is defined recursively by $f_0 = 0$, $f_1 = 1$, and $f_n = f_{n-1}f_{n-2}$ for $n \geq 2$. We denote $F_n = |f_n|$, where F_n satisfies the Fibonacci recurrence relation $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 1$ and $F_1 = 1$. We refer to the logical separation between f_{n-1} and f_{n-2}, in the string f_n, as the ***boundary***.

n	f_n	F_n
0	0	1
1	1	1
2	10	2
3	101	3
4	10110	5
5	10110101	8
6	1011010110110	13
7	101101011011010110101	21
8	1011010110110101101011011010110110	34

Table 5.1: f_n and F_n, for $0 \leq n \leq 8$

Bucci et al. [15] study the open and closed prefixes of Fibonacci words and show that the sequence of open and closed prefixes of a Fibonacci word follows the Fibonacci sequence. De Luca et al. [21] explore the sequence of open and closed prefixes of a Sturmian word. In [28], the authors define and find the closed Ziv–Lempel factorization and classify closed prefixes of infinite m-bonacci words.

For any integer $n \geq 2$, let $p_n = f_n[1..F_n - 2]$ denote the prefix of f_n excluding its last two symbols. Then, $f_n = p_n\,\delta_n$, where $\delta_n = 10$ if n is even and $\delta_n = 01$, otherwise. Let $P_n = F_n - 2$ denote the length of p_n. The golden ratio is defined as $\phi = \frac{1+\sqrt{5}}{2}$, and it is known that the ratio of consecutive Fibonacci numbers converges to ϕ, i.e., $\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = \phi$. Table 5.2 shows some of the initial p_n prefixes of Fibonacci words.

n	p_n	P_n
0	–	–
1	–	–
2	ε	0
3	1	1
4	101	3
5	101101	6
6	10110101101	11
7	1011010110110101101	19
8	10110101101101011010110110101101	32

Table 5.2: p_n and P_n, for $0 \leq n \leq 8$

The non-singleton MCSs are either runs or maximal gapped repeats in which the arm occurs exactly twice [9]. For a Fibonacci word f_n, let $SM(f_n)$, $R(f_n)$, and $GM(f_n)$ denote the number of singleton MCSs, runs, and gapped-MCSs, respectively (see Chapter 2 for definitions). Therefore, the total number of MCSs in f_n is given by $M(f_n) = SM(f_n) + R(f_n) + GM(f_n)$.

Lemma 5. *The no. of singleton MCSs in a Fibonacci word f_n, for $n \geq 4$, is:*

$$SM(f_n) = \begin{cases} F_{n-2} + F_{n-4} + 2, & \text{if } n \text{ is odd,} \\ F_{n-2} + F_{n-4}, & \text{if } n \text{ is even.} \end{cases} \tag{5.1}$$

Proof. We prove the Lemma by induction on n. The base cases, $SM(f_4) = 3$ and $SM(f_5) = 6$, match the direct counts.

For the inductive step, assume that Equation (5.1) holds for $k \geq 6$. By definition, $f_{k+1} = f_k f_{k-1}$. Every Fibonacci word f_k begins with a 10; if k is odd, it ends with 01 and if k is even, it ends with 10. By definition, the first and last character in f_k are singleton MCSs. Therefore, we examine whether the singleton MCSs at the boundary are retained in f_{k+1}.

Suppose $k+1$ is even. By inductive hypothesis, $SM(f_k) = F_{k-2} + F_{k-4} + 2$ and $SM(f_{k-1}) = F_{k-3} + F_{k-5}$. Since k is odd, f_k ends with 1 and f_{k-1} begins with 1, their concatenation results in the string 11 at the boundary, which forms a new non-singleton MCS, and reduces the count of the singleton MCSs in f_{k+1} by two. Thus, we get $SM(f_{k+1}) = SM(f_k) + SM(f_{k-1}) - 2 = F_{k-1} + F_{k-3}$.

Suppose $k+1$ is odd. By inductive hypothesis, $SM(f_k) = F_{k-2} + F_{k-4}$ and $SM(f_{k-1}) = F_{k-3} + F_{k-5} + 2$. Since k is even, f_k ends with 0 and f_{k-1} begins with 1, their concatenation results in the string 01 at the boundary, which does not reduce the count of singleton MCSs in f_{k+1}. Thus, we get $SM(f_{k+1}) = SM(f_k) + SM(f_{k-1}) = F_{k-1} + F_{k-3} + 2$. □

Lemma 6 (Theorem 2 in [32])**.** *For $n \geq 4$, the number of runs in the Fibonacci word f_n, is given by $R(f_n) = 2F_{n-2} - 3$.*

To compute $GM(f_n)$, we count the number of maximal gapped repeats whose arms occur exactly twice, as these are, by definition gapped-MCSs. In Theorem 5 we first show that the only gapped-MCSs in a Fibonacci word are occurrences of the substring 101 that are also maximal gapped repeats. Then, we count all such occurrences in f_n.

Lemma 7 (Theorem 1 in [48])**.** *Suppose uvu is a maximal gapped repeat in f_n that is also a suffix of f_n. Then, the arm u is a Fibonacci word.*

Lemma 8. *For $n \geq 5$, suppose uvu is a maximal gapped repeat in f_n that is also a suffix of f_n. Then uvu is not a gapped-MCS.*

Proof. By Lemma 7, the arm of the maximal gapped repeat is a Fibonacci word $u = f_k$. Clearly, $k \neq n$.

If n is even, then the proper suffixes of f_n that are Fibonacci words are in the set $\mathcal{S} = \{f_k \mid k \mod 2 = 0 \land 0 \leq k < n\}$. For any proper suffix $f_k \in \mathcal{S} \setminus \{f_0, f_2\}$, f_{k+2} is a suffix of f_n. By definition, $f_{k+2} = f_{k+1}f_k = f_kf_{k-1}f_k = f_kf_kf_{k-3}f_{k-2}$, the suffix $f_{k-1}f_k = f_kf_{k-3}f_{k-2}$ has length $F_k + F_{k-1} < 2F_k$ implying that the last two occurrences of f_k overlap. Therefore, any uvu with f_k as its arm, will have at least three occurrences of f_k making it an open string. For suffix $f_k \in \{f_0, f_2\}$ of f_n, for $n \geq 6$, $f_6 = 1011010110110$ is a suffix of f_n, and the last two occurrences of f_k are left-extendible, and so, it is not maximal. Moreover, any longer uvu with the arm f_k will have at least three occurrences of f_k making it an open string.

If n is odd, then $\mathcal{S} = \{f_k \mid k \mod 2 = 1 \land 1 \leq k < n\}$. The argument is analogous for $f_k \in \mathcal{S} \setminus \{f_1\}$ and $f_k \in \{f_1\}$, respectively. □

Lemma 9 (Theorem 2 in [48])**.** *For $n \geq 3$, suppose uvu is a maximal gapped repeat in f_n that is not a suffix of f_n. Then the arm u belongs to the set $\mathcal{A}_n = \{p_3, p_4, \ldots, p_{n-2}\}$.*

Lemma 10 (Lemma 8 in [48])**.** *For $n \geq 4$, all the borders of p_n form the set $\mathcal{B}_n = \{p_3, p_4, \ldots, p_{n-1}\}$.*

Lemma 11. *For $n \geq 5$, the border p_{n-1} of p_n is the longest proper cover of p_n.*

Proof. By Lemma 10, we know that p_{n-1} is the longest border of p_n. $F_n = F_{n-1}+F_{n-2}$. For $n \geq 5$, $F_n-4 = F_{n-1}+F_{n-2}-4$, and so $P_n-2 = P_{n-1}+P_{n-2}$. Hence, $P_n \leq 2 \cdot P_{n-1}$. Clearly, p_{n-1} is either an adjacent or an overlapping border of p_n. Therefore, p_{n-1} is the longest proper cover of p_n. □

Lemma 12 (Lemma 2 in [38])**.** *Let u be a proper cover of x and let $v \neq u$ be a substring of x such that $|v| \leq |u|$. Then, v is a cover of $x \Leftrightarrow v$ is a cover of u.*

Lemma 13. *For $n \geq 5$, all covers of p_n form the set $\mathcal{C}_n = \{p_4, p_5, \ldots, p_n\}$.*

Proof. By Lemma 11, for $n \geq 5$, p_{n-1} is the longest proper cover of p_n. Similarly, p_{n-2} is the longest proper cover of p_{n-1}, p_{n-3} is the longest proper cover of p_{n-2}, and so on. Therefore, by Lemma 12, it follows that all covers of p_n belong to the set $\mathcal{C}_n = \{p_4, p_5, \ldots, p_n\}$ for all $n \geq 5$. □

Theorem 5. *For $n \geq 5$, every gapped-MCS in a Fibonacci word f_n corresponds to an occurrence of the substring* 101 *that is also a maximal gapped repeat.*

Proof. All gapped-MCSs are maximal gapped repeats. By Lemma 8, for $n \geq 5$, all maximal gapped repeats that are also suffixes of f_n are not gapped-MCSs.

By Lemma 9, for $n \geq 3$, if uvu is a maximal gapped repeat in f_n that is not a suffix of f_n, then the arm u belongs to the set $\mathcal{A}_n = \{p_3, p_4, \ldots, p_{n-2}\}$. By Lemma 13, all covers of p_n belong to the set $\mathcal{C}_n = \{p_4, p_5, \ldots, p_n\}$.

Suppose there exists a maximal gapped $w = uvu$, where $u \in \mathcal{A}_n \setminus p_3$, that is a gapped-MCS. Assume n is odd. Then, $f_n = p_n\delta_n$, where $\delta_n = 01$, and $f_n[F_n-1] = 0$.

Since $u \in \mathcal{A}_n \setminus p_3$, it ends with 1. Clearly, u does not occur ending at position $F_n - 1$. Now suppose n is even. By a similar reasoning, u does not occur ending at $F_n - 1$ since it ends with 01 and $f_n[F_n - 2] = f_n[F_n - 1] = 1$ (as $f_4 = 10110$ is a suffix of f_n). Hence, u ends within p_n. Since u is a cover of p_n, $w = uvu$ will contain at least three occurrences of u making it an open string. Therefore, there is no maximal gapped repeat of the form $w = uvu$, where $u \in \mathcal{A}_n \setminus p_3$, that is a gapped MCS.

Now we consider the final case, where $u = p_3 = 1$. In this case, we get 101 as the only gapped-MCS. Any longer maximal gapped repeat will have at least three occurrences of $u = p_3 = 1$, making it an open string—a contradiction. □

Lemma 14. *The no. of gapped-MCSs in a Fibonacci word f_n, for $n \geq 5$, is:*

$$GM(f_n) = \begin{cases} F_{n-5}, & \text{if } n \text{ is odd,} \\ F_{n-5} + 1, & \text{if } n \text{ is even.} \end{cases} \tag{5.2}$$

Proof. We prove the Lemma by induction on n. By direct counting for the base cases, we verify that $GM(f_5) = 1$ and $GM(f_6) = 2$. For the inductive step, assume that Equation (5.2) holds for $k \geq 7$. For $n \geq 3$, $f_3 = 101$ is a prefix of every f_n, and by Theorem 5 it is the only gapped-MCS in f_n. In $f_{k+1} = f_k f_{k-1}$, any new occurrences of 101 in f_{k+1} can only occur beginning in f_k and ending in f_{k-1}. Moreover, an occurrence of 101 as a suffix of f_k or a prefix of f_{k-1} may be lost by extension. Below we examine all such cases.

Suppose $k + 1$ is odd. By inductive hypothesis, $GM(f_k) = F_{k-5} + 1$ and $GM(f_{k-1}) = F_{k-6}$. Moreover, f_k ends with the suffix $f_2 = 10$ and f_{k-1} begins with $f_3 = 101$, resulting in the formation of the string 101 crossing the boundary. Clearly, this string is not a gapped-MCS as it is right-extendible. The gapped-MCS

101 which is the prefix of f_{k-1} is no longer a gapped-MCS since f_k ends with a 0 making it left-extendible. Thus, we get $GM(f_{k+1}) = GM(f_k) + GM(f_{k-1}) - 1 = (F_{k-5} + 1) + F_{k-6} - 1 = F_{k-4}$.

Suppose $k+1$ is even. By inductive hypothesis, $GM(f_k) = F_{k-5}$ and $GM(f_{k-1}) = F_{k-6} + 1$. f_k ends with a 1, so the prefix 101 of f_{k-1} is retained as a gapped-MCS. By Lemma 8 the suffix $f_3 = 101$ of f_k is not a gapped-MCS. Thus, we get $GM(f_{k+1}) = GM(f_k) + GM(f_{k-1}) = F_{k-4} + 1$. □

Theorem 6. *The number of MCSs in a Fibonacci word f_n, denoted by $M(f_n)$, for $n \geq 5$, is given below in Equation (5.3). Furthermore, $M(f_n) \approx 1.382F_n$.*

$$M(f_n) = \begin{cases} F_n + F_{n-2} - 1, & \textit{if } n \textit{ is odd,} \\ F_n + F_{n-2} - 2, & \textit{if } n \textit{ is even.} \end{cases} \tag{5.3}$$

Proof. Adding Equations (5.1), (5.2) and the Equation in Lemma 6 and simplifying, we get Equation (5.3). Next, we have $M(f_n) < F_n + F_{n-2} = \left(1 + \frac{F_{n-2}}{F_n}\right) F_n \approx \left(1 + \frac{1}{\phi^2}\right) F_n \approx 1.382F_n$ (since $\lim_{n\to\infty} \frac{F_{n-2}}{F_n} = \frac{1}{\phi^2}$). □

Chapter 6

Implementation and Experiments

In this chapter, we introduce the first implementation of algorithms for computing closed strings and MCSs. This implementation enables us to assess the performance and practicality of the proposed method. We outline the experimental setup, the data structures and techniques employed, the process of string generation, and analyze the results.

6.1 Experimental Setup

All experiments were carried out on a server equipped with an Intel Xeon Gold 6426Y CPU (64 cores, 128 threads, 75 MiB L3 cache), 250 GiB of RAM and running Red Hat Enterprise Linux (RHEL) 9.5 with kernel version 5.14.0-503.26.1. The code was implemented in C++ and compiled using GCC 11.5.0. The implementation, along with a detailed README, is publicly available at `https://github.com/neerjamhaskar/closedStrings`.

6.2 Key Data Structures

The algorithm takes advantage of several essential data structures to efficiently compute the $\mathcal{MRC}$ array, as described below. These data structures collectively enable the proposed algorithm to achieve a time complexity of $\mathcal{O}(n \log n)$ for strings of length n, ensuring both time and space efficiency.

Suffix Array (SA) and Longest Common Prefix Array (LCP): The implementations for computing the suffix array (SA) and the longest common prefix array (LCP) were sourced from the *libsais* library [25], which is based on contributions from [30, 39, 45, 47].

The Extended Dis-joint Set: The *extended disjoint-set* [31] is a data structure designed to efficiently manage updates during the computation of the $\mathcal{MRC}$ array. Each set in this structure is represented as a height-balanced binary tree, such as an AVL tree or a red-black tree [17]. These trees ensure logarithmic upper bounds on their height, enabling efficient search and update operations.

To optimize operations, several auxiliary arrays are maintained. For each element $x \in \{1, \ldots, n\}$, the following arrays are stored:

- $\mathsf{next}[x] = next_{\mathcal{P}}(x)$: This provides the next element of x within its set.
- $\mathsf{tree}[x]$: A pointer to the tree representing the set P such that $x \in P$.

In practical implementation, the extended disjoint-set is realized using the C++ `std::set` container. This container, internally based on a red-black tree, supports key operations like insertion, `set::upper_bound`, and `set::lower_bound`, all with a time complexity of $\mathcal{O}(\log n)$. These functions are particularly useful for efficiently computing the *ChangeList*, which tracks updates to the successor and predecessor relationships between sets during *Union* operations.

The `set::insert` operation is used to merge elements into a set, while `set::upper_bound` and `set::lower_bound` facilitate determining and updating relationships within the merged sets. This combination of auxiliary arrays and efficient tree-based representation makes the extended disjoint set highly suitable for dynamic applications such as $\mathcal{MRC}$ computation.

Further details about the extended disjoint-set are provided in Section 2.4. For more information on the C++ `std::set` container, refer to the official documentation at https://en.cppreference.com/w/cpp/container/set.html.

Simple Stack: A simple stack is employed to store pointers to the suffix sets being processed at various stages of the algorithm. The stack allows efficient access to both the current and previously processed sets, enabling a minimal memory footprint. We utilize the template version of `std::stack` from the C++ library to ensure type safety and flexibility during implementation. For a complete documentation of `std::stack`, see the C++ documentation at https://en.cppreference.com/w/cpp/container/stack.html.

6.3 String Generation

Short Strings for Alphabets $\sigma = 2, 3, 4$: To generate strings over alphabets of size 2, 3, and 4, we utilize a simple integer counter, interpreting the counter value as numbers in base 2, 3, or 4. This method ensures that all possible strings of a given length are systematically enumerated without requiring pre-computation or storage. To parallelize the computation efficiently, we divide the task into buckets corresponding to the number of available threads. Each thread processes a distinct subset of numbers, dynamically generating the strings on-the-fly for computation. This approach not only leverages parallel processing to speed up the computation but also minimizes memory usage by avoiding the need to store large volumes of generated strings.

Fibonacci Words: For Fibonacci words, we adopt a file concatenation method to minimize main memory overhead. Starting with two base files containing the initial strings 0 and 1, we iteratively generate subsequent Fibonacci words by concatenating the contents of the previous two files. This incremental file-based approach avoids repeated memory loads and allows us to generate Fibonacci words of significant length efficiently.

6.4 Results

We computed the $\mathcal{MRC}$ array and the MCSs for a range of Fibonacci words of varying length. Analyzing the resulting structures and patterns provided valuable insights into their combinatorial properties. These observations were instrumental in deriving the theoretical results presented in Chapter 5, where the relationship between MRC arrays, MCSs, and specific features of Fibonacci words is explored in detail.

For $\sigma = 2, 3$ and 4, we generate all strings of length n for $1 \leq n \leq 20$ and compute their MCSs. The results are plotted in Figure 6.1. As observed, the maximum number of MCSs increases with the size of the alphabet σ for strings of the same length. In total, the number of strings analyzed is equal to $\sum_{i=1}^{20}(2^i + 3^i + 4^i) \approx 1.47$ trillion.

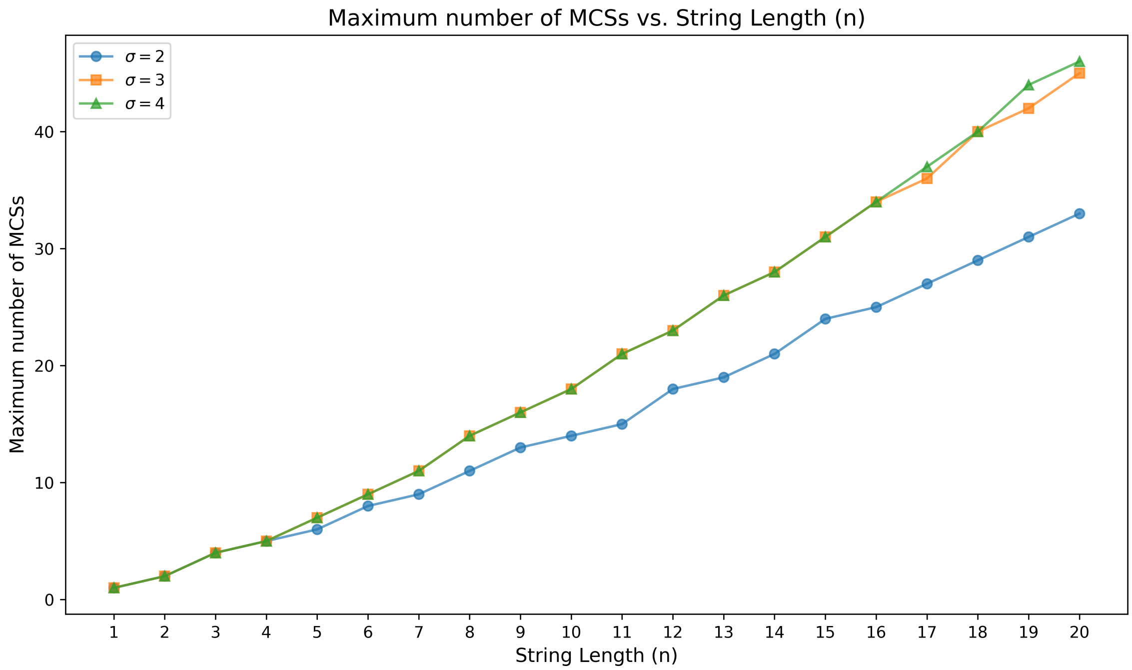

Figure 6.1: Maximum number of MCSs in strings of length n, where $1 \leq n \leq 20$, for alphabets of size $\sigma = 2, 3$ and 4 [29].

Through the identification of strings that achieve the maximum number of MCSs for small values of n and σ, alongside an analysis of Fibonacci words, we observed several interesting combinatorial patterns. These results provided valuable insights into the relationship between string structure and MCS properties, while also highlighting new open questions. These challenges, outlined in Chapter 7, offer directions for further exploration into the computational and theoretical aspects of closed substrings.

Chapter 7

Conclusion and Open Problems

7.1 Conclusion

This thesis introduced a compact representation for all $\mathcal{O}(n^2)$ closed substrings of a string $w[1..n]$, requiring only $\mathcal{O}(n \log n)$ space. The $\mathcal{MRC}$ array was introduced, along with an $\mathcal{O}(n \log n)$ time algorithm to compute it. Using this structure, efficient algorithms were developed to compute the compact representation of all closed substrings and identify all MCSs within the same time complexity. An exact formula for the number of MCSs in Fibonacci words was also derived, shedding light on their combinatorial properties. The practical implementation of these algorithms facilitates computation and discovery, bridging the gap between theoretical results and real-world applications. This contribution establishes a foundation for further exploration into the properties of closed substrings.

7.2 Open Problems

Several directions for future research remain open. One question is whether strings with the maximum number of MCSs can be generated algorithmically for arbitrary n and σ. Another is the possibility of deriving a closed-form formula or recurrence relation for their exact count. Furthermore, characterizing MCSs in m-bonacci words or more general Sturmian words presents a significant challenge. Understanding their structure and developing general formulas for their count in such sequences could provide deeper insights into their combinatorial and algorithmic properties.

Bibliography

[1] H. Alamro, M. Alzamel, C. S. Iliopoulos, S. P. Pissis, S. Watts, and W.-K. Sung. Efficient identification of k-closed strings. In G. Boracchi, L. Iliadis, C. Jayne, and A. Likas, editors, *Engineering Applications of Neural Networks*, pages 583–595, Cham, 2017. Springer International Publishing. doi: 10.1007/978-3-319-65172-9_49.

[2] M. Alzamel, C. S. Iliopoulos, W. Smyth, and W.-K. Sung. Off-line and on-line algorithms for closed string factorization. *Theoretical Computer Science*, 792: 12–19, 2019. ISSN 0304-3975. doi: 10.1016/j.tcs.2018.10.033. Special issue in honor of the 70th birthday of Prof. Wojciech Rytter.

[3] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993. doi: 10.1016/0304-3975(93)90159-Q.

[4] A. Apostolico, M. Farach, and C. S. Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991. doi: 10.1016/0020-0190(91)90056-N.

[5] P. Arnoux and G. Rauzy. Représentation géométrique de suites de complexité

$2n+1$. *Bulletin de la Société Mathématique de France*, 119(2):199–215, 1991. doi: 10.24033/bsmf.2164.

[6] G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto. Closed factorization. In J. Holub and J. Zda'rek, editors, *Proceedings of PSC 2014*, pages 162–168. Czech Technical University in Prague, Czech Republic, 2014. ISBN 978-80-01-05547-2. URL https://www.stringology.org/papers/PSC2014.pdf.

[7] G. Badkobeh, G. Fici, and Z. Lipták. On the number of closed factors in a word. In A.-H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications*, pages 381–390, Cham, 2015. Springer International Publishing. ISBN 978-3-319-15579-1. doi: 10.1007/978-3-319-15579-1_29.

[8] G. Badkobeh, H. Bannai, K. Goto, T. I, C. S. Iliopoulos, S. Inenaga, S. J. Puglisi, and S. Sugimoto. Closed factorization. *Discrete Applied Mathematics*, 212:23–29, 2016. ISSN 0166-218X. doi: 10.1016/j.dam.2016.04.009. Stringology Algorithms.

[9] G. Badkobeh, A. De Luca, G. Fici, and S. J. Puglisi. Maximal closed substrings. In D. Arroyuelo and B. Poblete, editors, *String Processing and Information Retrieval*, pages 16–23, Cham, 2022. Springer International Publishing. doi: 10.1007/978-3-031-20643-6_2.

[10] G. Badkobeh, A. D. Luca, G. Fici, and S. Puglisi. Maximal closed substrings, 2024.

[11] H. Bannai, S. Inenaga, T. Kociumaka, A. Lefebvre, J. Radoszewski, W. Rytter,

S. Sugimoto, and T. Waleń. Efficient algorithms for longest closed factor array. In *String Processing and Information Retrieval*, pages 95–102, Cham, 2015. Springer International Publishing. doi: 10.1007/978-3-319-23826-5_10.

[12] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In R. Giancarlo and D. Sankoff, editors, *Combinatorial Pattern Matching*, pages 397–411, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45123-5. doi: 10.1007/3-540-45123-4_33.

[13] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. In M. Crochemore and M. Paterson, editors, *Combinatorial Pattern Matching*, pages 134–149, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48452-3. doi: 10.1007/3-540-48452-3_11.

[14] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *J. ACM*, 26(2): 211–226, Apr. 1979. ISSN 0004-5411. doi: 10.1145/322123.322127.

[15] M. Bucci, A. De Luca, and G. Fici. Enumeration and structure of trapezoidal words. *Theoretical Computer Science*, 468:12–22, 2013. ISSN 0304-3975. doi: 10.1016/j.tcs.2012.11.007.

[16] M. Christou, C. S. Iliopoulos, and S. P. Pissis. On quasiperiodicities in fibonacci strings. *Theoretical Computer Science*, 640:78–85, 2016. doi: 10.1016/j.tcs.2016.05.013.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 4th edition, 2022. ISBN 9780262046305.

[18] M. Crochemore, G. M. Landau, and W.-K. Hon. On the number of maximal α-gapped repeats in strings. *Theoretical Computer Science*, 599:87–96, 2015.

[19] M. Crochemore, T. Lecroq, and H. Poulard. Shortest covers of all cyclic shifts of a string. *Journal of Computer and System Sciences*, 114:70–85, 2021. doi: 10.1016/j.jcss.2020.06.004.

[20] M. Crochemore, T. Lecroq, and W. Rytter. *Algorithms on Strings*. Cambridge University Press, Cambridge, England, July 2021. ISBN 9781108835831.

[21] A. De Luca, G. Fici, and L. Q. Zamboni. The sequence of open and closed prefixes of a sturmian word. *Advances in Applied Mathematics*, 90:27–45, 2017. ISSN 0196-8858. doi: 10.1016/j.aam.2017.04.007.

[22] G. Fici. A classification of trapezoidal words. In P. Ambrož, v. Holub, and Z. Masáková, editors, Proceedings 8th International Conference *Words 2011,* Prague, Czech Republic, 12-16th September 2011, volume 63 of *Electronic Proceedings in Theoretical Computer Science*, pages 129–137. Open Publishing Association, 2011. doi: 10.4204/EPTCS.63.18.

[23] G. Fici et al. Open and closed words. *Bulletin of EATCS*, 3(123), 2017.

[24] P. Gawrychowski, G. M. Landau, and L. Parida. Efficient algorithms for maximal α-gapped repeats. In *Proceedings of the 27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9724 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2016.

[25] I. Grebnov. libsais: A library for linear time suffix array, longest common prefix array and burrows wheeler transform construction based on induced sorting algorithm. GitHub repository, 2021–2025. URL https://github.com/IlyaGrebnov/libsais. Version 2.10.2, last updated June 12, 2025.

[26] C. S. Iliopoulos, D. W. G. Moore, and K. Park. Covering a string. *Algorithmica*, 16(4):288–297, 1996. doi: 10.1007/BF01955677.

[27] C. S. Iliopoulos, M. Matsoukas, and M. Mohamed. Covers of circular fibonacci strings. *Theoretical Computer Science*, 190(2):281–293, 1998. doi: 10.1016/S0304-3975(97)00216-7.

[28] M. Jahannia, M. Mohammad-Noori, N. Rampersad, and M. Stipulanti. Closed ziv–lempel factorization of the m-bonacci words. *Theoretical Computer Science*, 918:32–47, 2022. ISSN 0304-3975. doi: 10.1016/j.tcs.2022.03.019.

[29] S. K. Jain and N. Mhaskar. Efficient computation of closed substrings. In G. Badkobeh, J. Radoszewski, N. Tonellotto, and R. Baeza-Yates, editors, *String Processing and Information Retrieval*, pages 172–187, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-05228-5. doi: 10.1007/978-3-032-05228-5_15.

[30] J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In G. Kucherov and E. Ukkonen, editors, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-02441-2_17.

[31] T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Fast

algorithm for partial covers in words. *Algorithmica*, 73(1):217–233, Sept. 2015. ISSN 1432-0541. doi: 10.1007/s00453-014-9915-3.

[32] R. Kolpakov and G. Kucherov. On maximal repetitions in words. In G. Ciobanu and G. Păun, editors, *Fundamentals of Computation Theory*, pages 374–385, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48321-2. doi: 10.1007/3-540-48321-7_31.

[33] R. Kolpakov and G. Kucherov. Finding maximal repeats in a string in linear time. *Journal of Computational Biology*, 5(2):231–241, 2000.

[34] D. Kosolobov. Closed repeats, 2024.

[35] N. Mhaskar and W. Smyth. String covering: A survey. *Fundamenta Informaticae*, 190(1):17–45, Oct. 2023. ISSN 1875-8681. doi: 10.3233/fi-222164.

[36] N. Mhaskar and W. Smyth. String covering: A survey. *Fundamenta Informaticae*, 190(1):17–45, 2023. doi: 10.3233/FI-222164.

[37] T. Mieno, S. Takahashi, K. Seto, and T. Horiyama. Online and offline algorithms for counting distinct closed factors via sliding suffix trees. In R. Královič and V. Kůrková, editors, *SOFSEM 2025: Theory and Practice of Computer Science*, pages 172–183, Cham, 2025. Springer Nature Switzerland. doi: 10.1007/978-3-031-82697-9_13.

[38] D. Moore and W. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994. ISSN 0020-0190. doi: 10.1016/0020-0190(94)00045-X.

[39] G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, Oct 2011. doi: 10.1109/TC.2010.188.

[40] O. Parshina and S. Puzynina. Finite and infinite closed-rich words. *Theoretical Computer Science*, 984:114315, 2024. ISSN 0304-3975. doi: 10.1016/j.tcs.2023.114315.

[41] O. Parshina and L. Q. Zamboni. Open and closed factors in arnoux-rauzy words. *Advances in Applied Mathematics*, 107:22–31, 2019. ISSN 0196-8858. doi: 10.1016/j.aam.2019.02.007.

[42] J. Radoszewski and W. Zuba. Computing string covers in sublinear time. In *Proceedings of the 31st International Symposium on String Processing and Information Retrieval (SPIRE 2024)*, volume 14899 of *Lecture Notes in Computer Science*, pages 272–288, 2024. doi: 10.1007/978-3-031-72200-4_21.

[43] B. Smyth. *Computing Patterns in Strings*. Pearson Addison-Wesley, 2003. ISBN 9780201398397.

[44] Y. Tanimura, M. Takeda, and Y. Tabei. Efficient enumeration of maximal α-gapped repeats. In *Proceedings of the 26th International Symposium on Algorithms and Computation (ISAAC)*, volume 9472 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2015.

[45] N. Timoshevskaya and W.-c. Feng. Sais-opt: On the characterization and optimization of the sa-is algorithm for suffix array construction. In *2014 IEEE*

4th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS), pages 1–6, 2014. doi: 10.1109/ICCABS.2014.6863917.

[46] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE, 1973. doi: 10.1109/SWAT.1973.13.

[47] J. Y. Xie, G. Nong, B. Lao, and W. Xu. Scalable suffix sorting on a multicore machine. *IEEE Transactions on Computers*, 69(9):1364–1375, 2020. doi: 10.1109/TC.2020.2972546.

[48] K. Yamane, Y. Nakashima, K. Seto, and T. Horiyama. Maximal α-gapped repeats in a fibonacci string. In R. Královič and V. Kůrková, editors, *SOFSEM 2025: Theory and Practice of Computer Science*, pages 337–350, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-82697-9. doi: 10.1007/978-3-031-82697-9_25.