This electronic thesis or dissertation has been downloaded from the King's Research Portal at https://kclpure.kcl.ac.uk/portal/

Avoided Words, Overabundant Words, Maximal Palindromes and Applicatons

Gao, Jia

Awarding institution:
King's College London

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without proper acknowledgement.

END USER LICENCE AGREEMENT

Unless another licence is stated on the immediately following page this work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International licence. https://creativecommons.org/licenses/by-nc-nd/4.0/

You are free to copy, distribute and transmit the work

Under the following conditions:

- Attribution: You must attribute the work in the manner specified by the author (but not in any way that suggests that they endorse you or your use of the work).
- Non Commercial: You may not use this work for commercial purposes.
- No Derivative Works - You may not alter, transform, or build upon this work.

Any of these conditions can be waived if you receive permission from the author. Your fair dealings and other rights are in no way affected by the above.

Take down policy

If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. Jan. 2025

Avoided Words, Overabundant Words, Maximal Palindromes and Applications

Jia Gao

Department of Informatics

King's College London

This dissertation is submitted for the degree of

Doctor of Philosophy

Natural & Mathematical Sciences

September 2017

Abstract

The study of string algorithms is essential for computer science and computational molecular biology. Recently, a number of algorithms on strings appear in many fields, such as pattern matching, combinatorics on words, string processing and automata theory, because of their magnitude advances in applications and theories. These advances have made to develop faster algorithms and to deal with certain natural problems.

In this thesis, we focus on computing certain structures in biological sequences using different algorithmic methods. Firstly, we study the computation of avoided words and overabundant words in biological sequences. The observed frequency of the longest proper prefix, the longest proper suffix, and the longest infix of a word in a given sequence can be used for classifying that word as avoided or overabundant. This concept is particularly useful in DNA linguistic analysis. The definitions used for the expectation and deviation of the word in this statistical model were described and biologically justified by Brendel *et al*. [19]. We present some algorithms that can be used effectively for computing such words. Furthermore, experimental results, using both real and synthetic data, which further highlight the effectiveness of this model, show the efficiency and applicability of our implementation in biological sequence analysis.

Secondly, we consider a special type of uncertain sequence called weighted string. In a weighted string every position contains a subset of the alphabet and every letter of the alphabet is associated with a probability of occurrence such that the sum of probabilities at each position equals 1. We generalize Alatabbi *et al.*'s [2] solution for standard strings to compute maximal palindromes of a weighted string. We provide some efficient algorithms to compute maximal palindromes in weighted strings. Last but not least, we make available an implementation of our algorithms, using synthetic data, show the efficiency of our implementation.

Acknowledgements

I would like to dedicate this thesis to my loving parents who allowed me to make the realization of this thesis possible. Also, I would like to thank my family for their continuing support during these years.

I owe a great debt of gratitude to Professor Costas S. Iliopoulous and Professor Maxime Crochemore for their invaluable guidance and their teaching throughout these four years.

I am grateful to Dr. Solon Pissis and Dr. Manal Mohammed for introducing me to the field of their research and for their collaboration. Also, I wish to thank all the co-authors of my papers for their excellent cooperation.

I would like to thank all my colleagues and friends, who make my life as a Ph.D. student joyful.

Last but not least, thanks to my examiners, Dr. Dominique Revuz and Professor Prudence Wong for an intellectually stimulating and enjoyable viva.

Declaration

I hereby declare that except where explicitly stated otherwise in the text, this doctoral thesis was composed by myself and that the work contained therein is my own. The following articles were published during my period of research. Certain material and concepts from these publications will necessarily be presented within the body of this work.

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

2. Almirantis, Y., Charalampopoulos, P., Gao, J., Iliopoulos, C. S., Mohamed, M., Pissis, S. P., & Polychronopoulos, D. (2018). On overabundant words and their application to biological sequence analysis. Theoretical Computer Science.

3. Almirantis, Y., Charalampopoulos, P., Gao, J., Iliopoulos, C. S., Mohamed, M., Pissis, S. P., & Polychronopoulos, D. (2016, August). Optimal computation of avoided words. Algorithms in Bioinformatics: 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings. Frith, M. & Storm Pedersen, N. C. (eds.). Springer International Publishing Switzerland, p. 1-13.

4. Almirantis, Y., Charalampopoulos, P., Gao, J., Iliopoulos, C. S., Mohamed, M., Pissis, S. P., & Polychronopoulos, D. (2017). Optimal Computation of Overabundant Words. 17th International Workshop on Algorithms in Bioinformatics (WABI 2017). Schwartz, R. & Reinert, K. (eds.). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Vol. 88, p. 4:1-4:14 (Leibniz International Proceedings in Informatics (LIPIcs)).

5. Alzamel, M., Gao, J., Iliopoulos, C. S., Liu, C., & Pissis, S. P. (2017, August). Efficient computation of palindromes in sequences with uncertainties. Engineering Applications of Neural Networks: 18th International Conference, EANN 2017, Athens, Greece, August 25–27, 2017, Proceedings. Boracchi, G., Iliadis, L., Jayne, C. & Likas, A. (eds.). Cham: Springer International Publishing Switzerland, Vol. 744, p. 620-629.

Jia Gao

September 2017

Table of contents

List of figures x

List of tables xiii

1 Introduction 1

2 Basic Concepts 6

2.1 Strings 6

2.2 Finite Automata 7

2.3 Suffix Trie 9

2.4 Suffix Automation 11

2.5 Suffix Trees 11

2.6 Minimal Absent Words 17

2.7 Indexing Weighted Sequences 22

2.8 Molecular biology 27

3 Avoided words and Overabundant words 30

3.1 Background and Contributions 31

3.1.1 Background 31

3.1.2 Contributions 34
3.2 Preliminaries 37
3.2.1 Definitions and Notations 37
3.2.2 Tight Asymptotic Bounds on Minimal Absent Words 39
3.2.3 Useful Properties of Avoided Words 42
3.2.4 Useful Properties of Overabundant Words 44
3.3 Algorithms 51
3.3.1 Computation of Avoided words 51
3.3.2 Computation of All ρ-Avoided Words 58
3.3.3 Computation of All ρ-Overabundant words 64
3.4 Implementation and Experiments 68
3.4.1 Avoided words 68
3.4.2 Overabundant words 79
3.5 Conclusion 86

4 Maximal Palindromes **87**
4.1 Background and Contributions 88
4.1.1 Background 88
4.1.2 Contributions 91
4.2 Preliminaries 92
4.2.1 Definitions and Notations 92
4.2.2 Useful Properties of Maximal Palindromes 97
4.3 Algorithms 99
4.3.1 Computation of Smallest Maximal z-Palindromic Factorization 99

4.3.2 Computation of Longest z-Palindromic Array 111

4.4 Implementation and Experiments 115

4.4.1 Smallest Maximal z-Palindromic Factorization 115

4.4.2 Longest z-Palindromic Array 116

4.5 Conclusion 120

5 Conclusions and Future work **121**

References **123**

Appendix A Constructing the special-weighted strings **131**

List of figures

2.1 The transition diagram of a DFA accepting all strings over $\Sigma = \{0,1\}$ that have substring 01. [36] 8

2.2 Suffix trie of the string *ababbb*, $T(Suff(ababbb))$. With each final state (double circled) is associtated an output which is the location of the suffix in the string *ababbb* [22] 10

2.3 The suffix automaton, $S(ababbb)$, minimal automaton accepting the suffixes of the string *ababbb* [22] 11

2.4 The suffix tree $\mathcal{T}(x)$ for $x =$ `AGCGCGACGTCTGTGT`. Double-lined nodes represent terminal nodes labelled with the associated indices. The suffix-links for non-root internal nodes are dashed. 16

2.5 RNA stem-loops.(a) A schematic overview of an RNA stem-loop depicting the important parameters for the role of such a hairpin RNA. (b) The SECIS stem-loop structure element controlling selenoprotein synthesis. Right: A consensus of a secondary structure of a SECIS element. Left: A specific example of the SECIS element in Homo sapiens. [57] 29

3.1 The above figures illustrate the nodes (implicit or explicit) considered in a step (lines 6-36) of Algorithm OVERABUNDANTWORDS. The figure on the left presents the case where CHILD(v, α) is an internal node, while the right one the case that it is a leaf. Black nodes represent implicit nodes along the edge (v, q) that we have to consider as potential w_p, and the red dotted line joins them with the respective (white) explicit node that represents the longest suffix of this w_p, i.e. w_i. 65

3.2 Experiment I. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) data of length 1MB for variable k. 74

3.3 Experiment I. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) data of length 1MB for variable ρ. 75

3.4 Experiment II. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) data of length 1Mbp to 128Mbp. 76

3.5 Experiment III. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS using all chromosomes of the human genome. 77

3.6 Elapsed time of Algorithm OVERABUNDANTWORDS using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) sequences of length 1M to 128M. 83

4.1 Hairpins that are common to a set of closely-related sequences can be represented compactly as weighted strings. 90

4.2 Suffix tree of $x\#x^R\$ =$ `abaab#baaba$`; double-lined nodes represent terminal nodes labeled with the associated indices. 98

4.3 The WI for X and $1/z$ shown in Example 4.3.1 (labels of edges to terminal nodes are appended with a letter $\$ \notin \Sigma$ for convenience). . . . 101

4.4 The graph $\mathcal{G}_X$ for X and $1/z$ shown in Example 4.3.2. 110

4.5 $\mathcal{PA}$ for X and $1/z$, this graph shown in Example 4.3.3. 114

4.6 Experiment 1. Elapsed time and maximal memory usage of Algorithm SMALLEST MAXIMAL Z-PALINDROMIC FACTORIZATION using synthetic DNA ($\sigma = 4$) data of length 1MB for variable z. 117

4.7 Experiment 2. Elapsed time and maximal memory usage of Algorithm SMALLEST MAXIMAL Z-PALINDROMIC FACTORIZATION using synthetic DNA ($\sigma = 4$) data of length 250KB to 4000KB. 118

4.8 Experiment 3. Elapsed time and maximal memory usage of Algorithm LONGEST Z-PALINDROMIC ARRAY using synthetic DNA ($\sigma = 4$) data of length 250KB to 4000KB. 119

A.1 The WI for X and $1/z$ in Example A.0.1 (labels of edges to terminal nodes are appended with a letter for convenience). 133

List of tables

3.1 The number of avoided words, for $k = 10$ and $\rho = -2$, for each concatenate of surrogates (Row 1); the number of avoided words of the corresponding CNE dataset (Row 2); and their ratio (Row 3). 78

3.2 The number of avoided words, for $k > 2$ and $\rho = -2$, for each concatenate of surrogates (Row 1); the number of avoided words of the corresponding CNE dataset (Row 2); and their ratio (Row 3). 78

3.3 The deviation of the randomly generated inserted word w, as well as the word $w_{\max}$ with the maximum deviation. The length of each of the 25 randomly generated sequences over $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ was $n = 80,000$, the length of w was $m = 6$, and $\rho = 0.000001$. In green are the cases when the word with the maximum deviation was w itself or one of its factors. 84

3.4 Number of overabundant words for $k = 10$ and $\rho = 3$. 85

3.5 Number of overabundant words for $k > 2$ and $\rho = 3$. 85

4.1 Computing odd-length maximal palindromes of $x = \mathtt{abaab}$ using the suffix tree of $x\#x^R\$ = \mathtt{abaab\#baaba\$}$. 97

4.2 Computing even-length maximal palindromes of $x = \texttt{abaab}$ using the suffix tree of $x\#x^R\$ = \texttt{abaab\#baaba\$}$. 98

4.3 Computing $\mathcal{F}$ and $\mathcal{U}$ from $\mathcal{MP}(X,z)$ for X and $1/z$ shown in Example 4.3.2. 110

4.4 Computing $\mathcal{F}$, $\mathcal{U}$ and $\mathcal{PA}$ from $\mathcal{MP}(X,z)$ for X and $1/z$ shown in Example 4.3.3. 114

Chapter 1

Introduction

The study of string algorithms is essential for computer science and computational molecular biology. Recently, a number of algorithms on strings appear in many fields, such as pattern matching, combinatorics on words, string processing and automata theory, because of their magnitude advances in applications and theories. These advances have made to develop faster algorithms and to deal with certain natural problems. In this thesis, we focus on computing certain structures in biological sequences using different algorithmic methods.

The observed frequency of the longest proper prefix, the longest proper suffix, and the longest infix of a word w in a given sequence x can be used for classifying w as avoided or overabundant. This concept is particularly useful in DNA linguistic analysis. The definitions used for the expectation and deviation of w in this statistical model were described and biologically justified by Brendel *et al.* [19]. The value of the deviation of w, denoted by $dev(w)$. A word w of length $k > 2$ is a ρ-avoided word in x if $dev(w) \leq \rho$ for a given threshold $\rho < 0$, or a ρ-overabundant word in x if $dev(w) \geq \rho$ for a given

threshold $\rho > 0$. Notice that such a word may be completely *absent* from x. Hence, computing all such words naïvely can be a very time-consuming procedure, in particular for large k.

We provide an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm to compute all ρ-avoided words of length k in a sequence of length n over a fixed-sized alphabet. For integer alphabets, our algorithm runs in time $\mathcal{O}(\sigma n)$ and is optimal for a sufficiently large alphabet of size σ. We also provide a time-optimal $\mathcal{O}(\sigma n)$-time algorithm to compute all ρ-avoided words (of any length) in a sequence of length n over an integer alphabet. Moreover, we provide a tight asymptotic upper bound for the number of ρ-avoided words over an integer alphabet and the expected length of the longest one. In the process, we show that the known asymptotic upper bound on the number of minimal absent words of a sequence is tight for integer alphabets. We also show that the same asymptotic bound is tight for the number of minimal absent words of a fixed length if the alphabet is sufficiently large.

We extend this study by presenting an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all overabundant words in a sequence x of length n over an integer alphabet. Our main result is based on a new *non-trivial* combinatorial property of the suffix tree $\mathcal{T}$ of x: the number of distinct factors of x whose longest infix is the label of an explicit node (which is the internal node of $\mathcal{T}$ holds more than one child node) of $\mathcal{T}$ is no more than $3n-4$. We further show that the presented algorithm is time-optimal by proving that $\mathcal{O}(n)$ is a tight upper bound for the number of overabundant words.

In Chapter 4, we consider a special type of uncertain sequence called weighted string. In a weighted string every position contains a subset of the alphabet and every

letter of the alphabet is associated with a probability of occurrence such that the sum of probabilities at each position equals 1. Usually a cumulative weight threshold $1/z$ is specified, and one considers only strings that match the weighted string with probability at least $1/z$. We generalize Alatabbi *et al.*'s [2] solution for standard strings to compute maximal palindromes of a weighted string.

We provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm, where n is the length of the weighted string and $1/z$ is the given threshold, to compute a smallest maximal z-palindromic factorization of a weighted string. Along the way, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute all maximal z-palindromes in weighted strings. We also provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute a longest z-palindromes array, which presents the longest length of a maximal palindrome ending at each position.

Thesis Structure

In Chapter 2 we specify several basic concepts and notations on strings that are utilized to represent most definitions and problems on strings. And then, we introduce the finite automata, suffix trie, suffix automation, also we introduce the specific data structure called suffix trees that is useful to design string algorithms and analyze their performance. At last, we introduce the minimal absent words and indexing weighted sequences, also the background of molecular biology.

In Chapter 3 we study the computation of avoided words and overabundant words in biological sequences. We present some algorithms that can be used effectively for computing such words, for instance, we present an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm to compute all ρ-avoided words of length k in a given sequence of length n over a fixed-sized alphabet, we also present an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm to compute all overabundant words in a sequence x of length n over an integer alphabet. Furthermore, experimental results, using both real and synthetic data, which further highlight the effectiveness of this model, show the efficiency and applicability of our implementation in biological sequence analysis.

In Chapter 4 we consider a special type of uncertain sequence called weighted string. We provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm, where n is the length of the weighted string and $1/z$ is the given threshold, to compute a smallest maximal z-palindromic factorization of a weighted string. Along the way, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute all maximal z-palindromes in weighted

strings. Moreover, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute a longest z-palindromes array in weighted strings. Last but not least, we make available an implementation of our algorithms, using synthetic data, show the efficiency of our implementation.

The final chapter (Chapter 5) concludes the thesis.

Chapter 2

Basic Concepts

In this chapter, firstly, we specify several basic concepts and notations on strings that are utilized to represent most definitions and problems on strings. And next, we introduce the finite automata, suffix trie, suffix automation, also we introduce the specific data structure called suffix trees that is useful to design string algorithms and analyze their performance. We will specify most definitions at the point where they are encountered, but several concepts and notation are so basic that we specify them at this chapter. And then, we provide a introduction to the minimal absent words and indexing weighted sequences, and a brief introduction to the molecular biology, as an attempt to acquaint ourselves with the most basic concepts.

2.1 Strings

We begin with basic definitions and notations generally following [22]. A finite, nonempty set of symbols is called an *alphabet*, that is denoted by Σ. Also those symbols are called *letters*. And the *size* of alphabet Σ is denoted by $\sigma = |\Sigma|$. A *string*

(sometimes called *word*) over Σ is a finite sequence of letters of an ordered alphabet Σ. The *length* of a finite string $x = x[0]x[1]\,..\,x[n-1]$ is denoted by $n = |x|$. Note that the zero length string (or word) is called *empty string* (sometimes called *empty word*), and it is denoted by ε. In what follows we assume without loss of generality that $\Sigma = \{0, 1, \ldots, \sigma - 1\}$. We also define Σ_x to be the alphabet of word x and $\sigma_x = |\Sigma_x|$.

For two positions i and j on x, we denote by $x[i\,..\,j] = x[i]\,..\,x[j]$, and the *factor* of x that starts at position i and ends at position j (it is empty if $j < i$). We call that a prefix of x is a factor that starts at position 0 ($x[0\,..\,j]$) and a suffix is a factor that ends at position $n-1$ ($x[i\,..\,n-1]$), and that a factor of x is a *proper* factor if it is not x itself. A factor of x that is neither a prefix nor a suffix of x is called an *infix* of x.

Let w be a word of length m, $0 < m \leq n$. We say that there exists an *occurrence* of w in x, or, more simply, that *w occurs in x*, when w is a factor of x. Every occurrence of w can be characterised by a starting position in x. Thus we say that w occurs at the *starting position* i in x when $w = x[i\,..\,i+m-1]$.

2.2 Finite Automata

A *deterministic finite automaton*, (DFA), is of the fundamental potential models of calculation. [36] That can be considered as *acceptor*, in other words, given strings as input and are either rejected or accepted.

A DFA starts in an *initial state*, and after reading the input that state can be in one of a finite number of states. On account of the symbols of string w, the DFA takes as input a string w. The symbols read in order from left to right, and the DFA moves from state to state. After reading all the symbols of string w, and then, if the DFA is in

a important state, namely an *accepting* state, or more commonly, *final state*, then we consider that string is accepted; otherwise, that string is rejected. The DFA accept the language which is the set of all accepted strings.

A directed graph could represent a DFA, namely a *transition diagram*. The new state of the machine can be indicated by a directed edge labeled with a letter when we read the given letter. Accordingly, the initial state can be drawn with an unlabeled arrow accessing the state, and double circles draw accepting states .

For example, a DFA is presented in Fig 2.1, that presents the transition diagram of a DFA accepting all string over $\Sigma = \{0, 1\}$ with a substring 01.

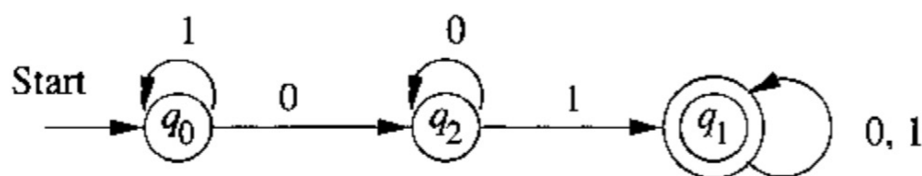


Fig. 2.1 The transition diagram of a DFA accepting all strings over $\Sigma = \{0, 1\}$ *that have substring* 01. *[36]*

More formally, a DFA can be represented as a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

Q is a finite set of states,

Σ is the finite input alphabet,

δ: $Q \times \Sigma \to Q$ is the *transition function*,

$q_0 \in Q$ is the *initial state*,

$F \subseteq Q$ is the set of accepting states.

Note that a DFA is supposed to be *complete*, δ is denoted for all pairs in its range. for the sake of formally denote acceptance by a DFA, we should to increase the domain of δ to $Q \times \Sigma^*$. First, we denote $\delta(q,\varepsilon) = q$ for all $q \in Q$, and denote $\delta(q,ba) = \delta(\delta(q,b,a)$ for all $q \in Q$, $b \in \Sigma^*$, and $a \in \Sigma$. Then $L(M)$, that M accept the language, denote to be

$$L(M) = \{w \in \Sigma^* : \delta(q_0,w) \in F\}$$

We name a state q of a DFA *reachable* if there exists $b \in \Sigma^*$ such that $\delta(q_0,b) = q$, and *unreachable* otherwise. Obviously, unreachable states could be deleted without altering the language accepted by a DFA.

2.3 Suffix Trie

The *suffix trie* of a string is the deterministic automaton, that cognises the set of suffixes of the string, where two distinct paths of the same source frequently have clear ends. Consequently, the underlying graph structure of the automaton is a tree, and letters label the tree's arcs. [22]

Regarding a tree shows that the recognized language's strings one to one map the tree's terminal or final states. If its language is finite, then the tree is so. As a result, an algorithm interests the explicit representation of that tree only for finite languages. [22]

At times one assesses trees, on external nodes of tree or leaves, to only have final states. For this reason, if no suitable prefix of a string of L is in L, and then a tree represents a language L. It brings about this statement that if x is a nonempty string, only $Suff(x) \setminus \{\varepsilon\}$ is expressible by a tree keeping this property, and this only occurs when the x's last letter appears only once in x. This is the real reason why one adds a particular letter at the end of the string at whiles. We appoint an output to nodes of the

trie that matches better with the automaton's concept. Terminal nodes consider some nodes whose output is denoted. [22]

The suffix trie's nodes are the factors of x, ε is the initial state, and the suffixes of x are the final states. If ua is a factor of x and $a \in A$, and then $\delta(u,a) = ua$ denote the transition function δ of $T(Suff(x))$. The output of a final state, which is then a suffix, is the location of this suffix in x. Accordingly, the string's length allocate the initial state or the root as output. Fig 2.2 shows an example of automaton. [22]

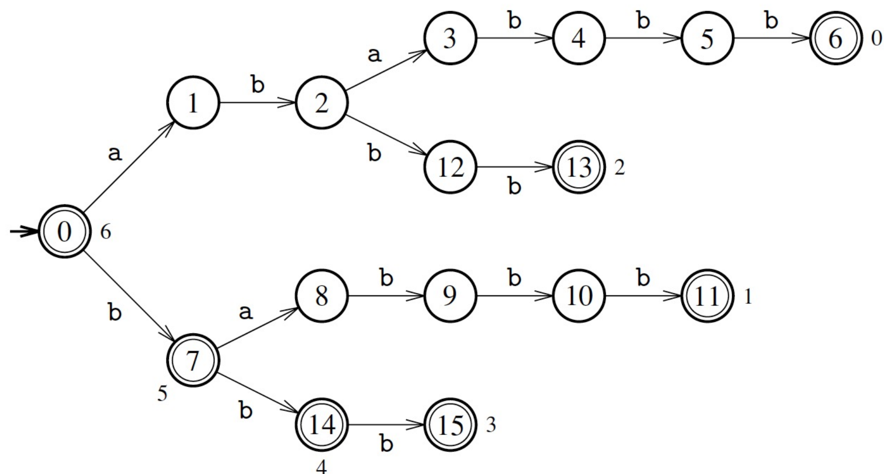


Fig. 2.2 Suffix trie of the string ababbb, $T(Suff(ababbb))$. With each final state (double circled) is associtated an output which is the location of the suffix in the string ababbb [22]

2.4 Suffix Automation

The *suffix automaton* of a string X, denoted by $S(x)$. That is the minimal automaton which accepts the set of suffixes of x. The most astonishing property is that automaton is, in the length of x, that its size is linear, although the number of factors of x could be quadratic. On a fixed alphabet, the construction of that automaton holds a linear time. Fig 2.3 exhibits a case of such automaton. [22]

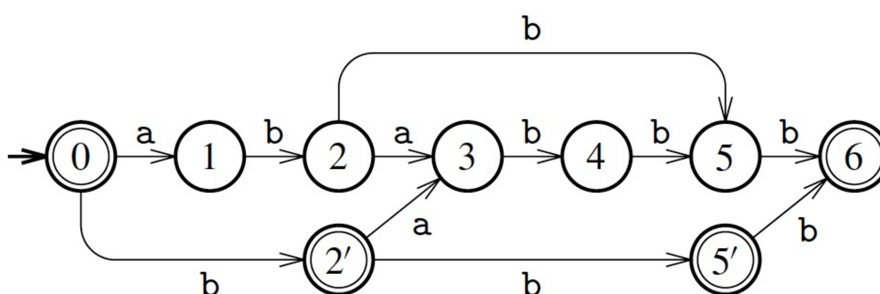


Fig. 2.3 The suffix automaton, $S(ababbb)$, minimal automaton accepting the suffixes of the string ababbb [22]

2.5 Suffix Trees

In our algorithms, suffix trees are used extensively as computational tools. For a general introduction to suffix trees, see [22].

The *suffix tree* $\mathcal{T}(x)$ of a non-empty word x of length n is a compact trie representing all suffixes of x. The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Then, each node of the trie

can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path.

We use $\mathcal{L}(v)$ to denote the *path-label* of a node v, i.e., the concatenation of the edge labels along the path from the root to v. We say that v is path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *word-depth* of node v. Node v is a *terminal* node, if and only if, $\mathcal{L}(v) = x[i \mathinner{.\,.} n-1]$, $0 \leq i < n$; here v is also labelled with index i. It should be clear that each occurring word w in x is uniquely represented by either an explicit or an implicit node of $\mathcal{T}(x)$. The *suffix-link* of a node v with path-label $\mathcal{L}(v) = \alpha w$ is a pointer to the node path-labelled w, where $\alpha \in \Sigma$ is a single letter and w is a word. The suffix-link of v exists if v is a non-root internal node of $\mathcal{T}(x)$. We denote by $\textsc{Child}(v, \alpha)$ the explicit node that is obtained from v by traversing the outgoing edge whose label starts with $\alpha \in \Sigma$. It is well-known that the suffix tree of x including the suffix-links can be computed in time and space $\mathcal{O}(n)$ [26]. Then all occurrences of a pattern of length m can be found in time $\mathcal{O}(m + Occ)$, where Occ is the number of occurrences [33]. It can also be preprocessed in time and space $\mathcal{O}(n)$ so that *lowest common ancestor* (LCA) queries for any pair of explicit nodes can be answered in $\mathcal{O}(1)$ time per query [18].

In any standard implementation of the suffix tree, we assume that each node of the suffix tree is able to access its parent. Note that once $\mathcal{T}(x)$ is constructed, it can be traversed in a depth-first manner to compute the word-depth $\mathcal{D}(v)$ for each node v. Let u be the parent of v. Then the word-depth $\mathcal{D}(v)$ is computed by adding $\mathcal{D}(u)$ to the length of the label of edge (u, v). If v is the root then $\mathcal{D}(v) = 0$. Additionally, a depth-first traversal of $\mathcal{T}(x)$ allows us to count, for each node v, the number of terminal

nodes in the subtree rooted at v, denoted by $\mathcal{C}(v)$, as follows. When internal node v is visited, $\mathcal{C}(v)$ is computed by adding up $\mathcal{C}(u)$ of all the nodes u, such that u is a child of v, and then $\mathcal{C}(v)$ is incremented by 1 if v itself is a terminal node. If a node v is a leaf then $\mathcal{C}(v) = 1$.

We assume that the terminal nodes of $\mathcal{T}(x)$ have suffix-links as well. We can either store them while building $\mathcal{T}(x)$ or just traverse it once and construct an array $node[0\,..\,n-1]$ such that $node[i] = v$ if $\mathcal{L}(v) = x[i\,..\,n-1]$. We further denote by PARENT(v) the parent of a node v in $\mathcal{T}(x)$ and by CHILD(v, α) the explicit node that is obtained from v by traversing the outgoing edge whose label starts with $\alpha \in \Sigma$.

Example 2.5.1. Consider the word `AGCGCGACGTCTGTGT`. Fig. 2.4 represents the suffix tree $\mathcal{T}(x)$. Note that word `GCG` is represented by the explicit node v; whereas word `TCT` is represented by the implicit node along the edge connecting the node labelled 15 and the node labelled 9. Consider node v in $\mathcal{T}(x)$; we have that $\mathcal{L}(v) =$ `GCG`, $\mathcal{D}(v) = 3$, and $\mathcal{C}(v) = 2$.

Theorem 2.5.1 ([22])**.** The operation SUFFIX-TREE(x,n), that produces $\mathcal{T}(x)$, takes a time $\mathcal{O}(n \times \log \text{card A})$ in the comparison model.

```
1  Algorithm SUFFIX-TREE(x,n) [22]
2      M ← NEW-AUTOMATON();
3      sℓ[initial[M]] ← initial[M];
4      (fork,k) ← (initial[M],0);
5      for i ← 0 to n − 1
6          k ← max{k,i};
7          if sℓ[fork] = NIL
8              t ←parent of fork;
9              Interval[proxa][0..σ − 1] ← 0;
10             if t = initial[M]
11                 ℓ ← ℓ − 1;
12             sℓ ← FAST-FIND(sℓ[t],k − ℓ,k);
13         (fork,k) ← SLOW-FIND(sℓ[fork],k);
14         if k < n
15             q ← NEW-STATE();
16             Succ[fork] ← Succ[fork] ∪ {((k,n − k),q)};
17         else q ← fork;
18         output[q] ← i;
19     output[initial[M]] ← n;
20     return M;
```

```
1  Algorithm SLOW-FIND(p,n) [22]
2      while k < n and TARGET(p,x[k]) ≠ NIL
3          q ← TARGET(p,x[k]);
4          (j,ℓ) ← label(p,q);
5          i ← j;
6          do  i ← i+1;
7              k ← k+1;
8          while i < j+ℓ and k < n and x[i] = x[k];
9          if i < j+ℓ;
10             Succ[p] ← Succ[p] \ {((j,ℓ),q)};
11             r ← NEW-STATE;
12             Succ[p] ← Succ[p] \ {((j,i−j),r)};
13             Succ[r] ← Succ[r] \ {((i,ℓ−i−j),q)};
14             return(r,k);
15         p ← q;
16     return(p,k);
```

```
1  Algorithm FAST-FIND(r, j,k) [22]
2      ▷ Computation of δ(r,x[j..k−1]);
3      if j ≥ k;
4          return r;
5      else q ← TARGET(r,x[j]);
6          if j+ℓ ≤ k;
7              return FAST-FIND(q, j+ℓ,k);
8          else Succ[r] ← Succ[r] \ {((j′,ℓ),q)};
9              p ← NEW-STATE();
10             Succ[r] ← Succ[r] \ {((j′,k−j),p)};
11             Succ[p] ← Succ[p] \ {((j′+k−j,ℓ−k+j),q)};
12             return p;
```

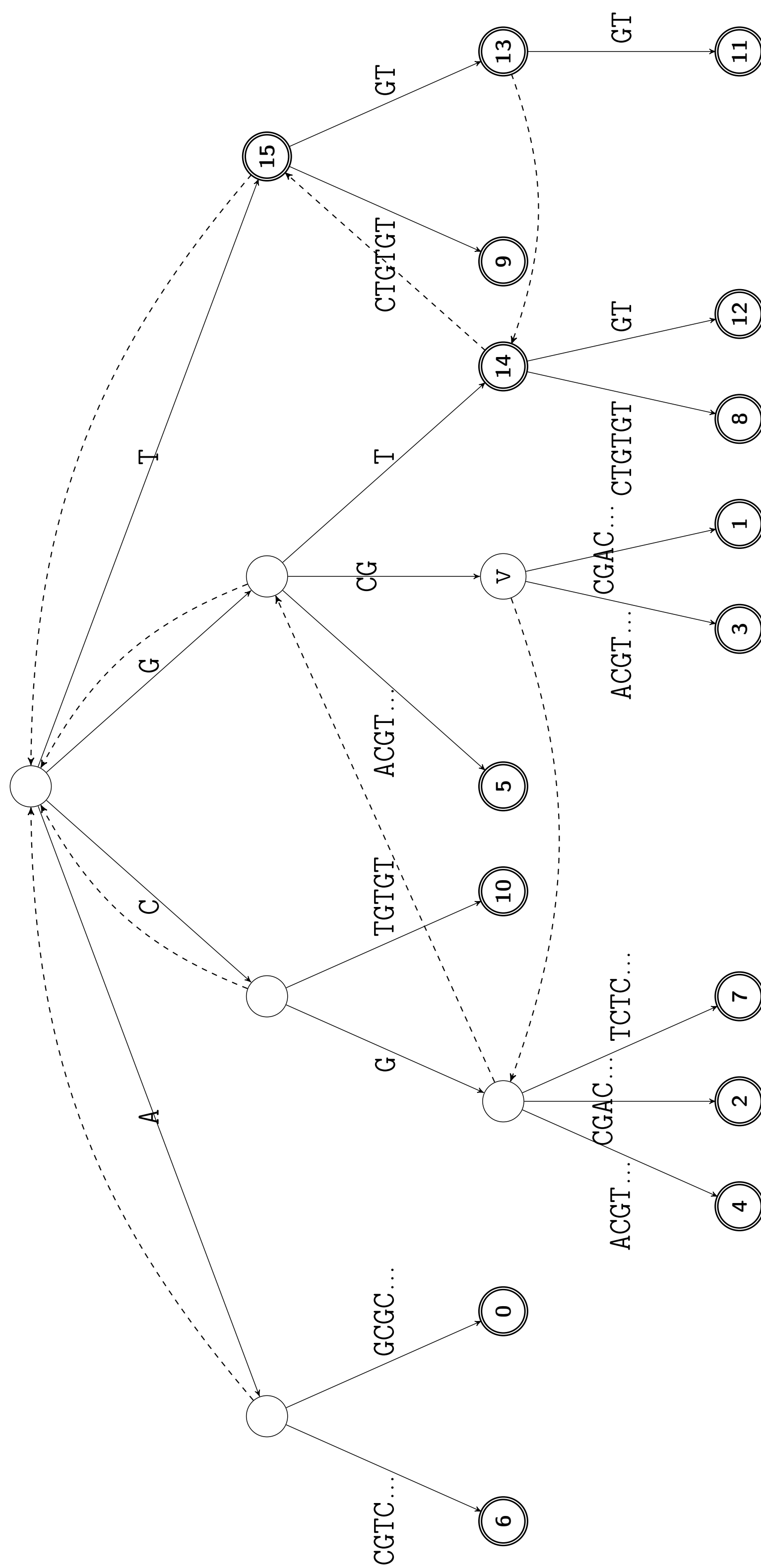


Fig. 2.4 The suffix tree $\mathcal{T}(x)$ for $x =$ AGCGCGACGTCTGTGT. Double-lined nodes represent terminal nodes labelled with the associated indices. The suffix-links for non-root internal nodes are dashed.

2.6 Minimal Absent Words

We define that the word w with length m is an *absent word* of string x with length n if it does not appear in string x. The absent word w, the length of such word is more than 2, of x is *minimal* if and only if all its proper factors appear in x. We also define by SA the *suffix array* of x, that is the array of length $|x|$ of the starting locations of all sorted suffixes of x, for all $1 \leq r < n$, we hold $x[SA[r-1]\,..\,n-1] < x[SA[r]\,..\,n-1]$ [44]. And $LCP(r,s)$ define the length of the longest common prefix of the words $x[SA[r]\,..\,n-1]$ and $x[SA[s]\,..\,n-1]$, for all $0 \leq r,\ s < n$, and 0 otherwise. We define by LCP the *longest common prefix* array of x denoted by $LCP[r] = lcp(r-1,r)$, for all $1 \leq r < n$, and $LCP[0] = 0$. The inverse iSA of the array SA is denoted by $iSA[SA[r]] = r$, for all $0 \leq r < n$. SA [49], iSA, and LCP [28] of x can be calculated in linear time and space. [10]

MINIMALABSENTWORDS
Input: A word x on Σ of length n
Output: For every minimal absent word w of x, one tuple $< a,(i,j) >$, such that w is defined by $w[0] = a$, $a \in \Sigma$, and $w[1\,..\,m-1] = x[i\,..\,j]$, $m \geq 2$.

Now we present MAW algorithm [10], which is a linear time and space algorithm for finding all minimal absent words in a word of length n using arrays SA and LCP.

The idea of algorithm MAW is to see at the occurrences of a factor w' of x, and at the letters that precede and follow these occurrences. If a couple (a,b), $a,b \in \Sigma$, can be

found, such that $aw^{'}$ and $w^{'}b$ appear in x, but $aw^{'}b$ does not appear in x, then we can think that $aw^{'}b$ is a minimal absent word of x.

A minimal absent words $w[0\,..\,m-1]$ of a word $x[0\,..\,n-1]$ is an absent word whose proper factors all appear in x. And then, $w_1 = w[1\,..\,m-1]$ and $w_2 = w[1\,..\,m-2] = w_1[0\,..\,|w_1|-2]$ appear in x; we now consider these two factors to characterise the minimal absent words. We focus each occurrence of w_1 and w_2, and create the sets of letters that appear just before:

$B_1(w_1) = \{x[j-1] : j$ is the starting position of an occurrence of $w_1\}$

$B_2(w_1) = \{x[j-1] : j$ is the starting position of an occurrence of $w_1[0\,..\,|w_1|-2]\}$

Lemma 2.6.1 ([10])**.** Let w and x be two words. Then w is a minimal absent word of x if $w[0]$ is an element of $B_2(w_1)$ and not of $B_1(w_1)$, with $x_1 = x_[1\,..\,m-1]$.

Lemma 2.6.2 ([10])**.** Let w be a minimal absent word of length m of word x of length n. Then there exists an integer $i \in [0 : n-1]$ such that $x[SA[i]\,..\,SA[i]+LCP[i]] = w_1$ or $x[SA[i]\,..\,SA[i]+LCP[i+1]] = w_1$, where $w_1 = w[1\,..\,m-1]$.

By Lemma 2.6.2, we can find all minimal absent words of x by checking only the factors $S_{2i} = x[SA[i]\,..\,SA[i]+LCP[i]]$ and $S_{2i+1} = x[SA[i]\,..\,SA[i]+LCP[i+1]]$, for all i in $[0 : n-1]$. We create the sets $B_1(S_{2i})$, $B_2(S_{2i})$ and $B_1(S_{2i+1})$, $B_2(S_{2i+1})$, where $B_1(S_j)$ (resp. $B_2(S_j)$) is the set of letters that directly precede an occurrence of the factor S_j (resp. the longest proper prefix of S_j), for all j in $[0 : 2n-1]$. And then, by Lemma 2.6.1, the difference between $B_2(S_j)$ and $B_1(S_j)$, for all j in $[0 : 2n-1]$, shows us all the minimal absent words of x. [10]

Consequently, the crucial computational step is to find these sets of letters efficiently. Now we visit twice arrays SA and LCP utilizing another array defined by B_1 (resp.

B_2) to hold set $B_1(S_j)$ (resp. $B_2(S_j)$), for all j in $[0 : 2n-1]$. Bothe arrays B_1 and B_2 compose of $2n$ elements, where each element is a bit vector of length σ, the size of the alphabet, according to one bit per alphabet letter. While iteration over arrays *SA* and *LCP*, we keep another array defined by $Interval$, such that, at the end of each iteration i, the ℓth element of $Interval$ holds the set of letters we have met before the prefix of length ℓ of $x[SA[i]\,..\,n-1]$. Array $Interval$ composes of $max_{i \in [0:n-1]} LCP[i]+1$ elements, where each element is a bit vector of length σ. [10]

In the first pass, we go arrays *SA* and *LCP* from top to bottom. For each $i \in [0 : n-1]$, we hold in locations $2i$ and $2i+1$ of B_1 (resp. B_2) the set of letters that directly precede occurrences of S_{2i} and S_{2i+1} (resp. their longest proper prefixes) whose starting locations occur before location i in *SA*. In the second pass, we visit bottom up to accomplish the sets, which are already stored, with the letters preceding the occurrences whose starting locations occur after location i in *SA*. For be efficient, we keep a structure of stack, defined by $LifoLCP$, to keep the *LCP* values of the factors that are prefixes of the one we are visiting at the moment. [10]

Theorem 2.6.3 ([10])**.** Algorithm MAW solves Problem MINIMALABSENTWORDS in time and space $\mathcal{O}(n)$.

```
1  Algorithm Top-Down-Pass(x, n, SA, LCP, B_1, B_2, σ) [10]
2      Interval[0 .. max_{i∈[0:n−1]} LCP[i]][0 .. σ − 1] ← 0;
3      LifoLCP.push(0);
4      foreach i ∈ [0 : n − 1] do
5          if i > 0 and LCP[i] < LCP[i − 1]
6              while LifoLCP.top() > LCP[i]
7                  proxa ← LifoLCP.pop();
8                  Interval[proxa][0 .. σ − 1] ← 0;
9              if LifoLCP.top() < LCP[i]
10                 Interval[LCP[i]] ← Interval[proxa];
11             B_1[2i − 1] ← Interval[proxa];
12             B_2[2i − 1] ← Interval[LCP[i]];
13         if SA[i] > 0
14             u ← x[SA[i] − 1];
15             value ← LifoLCP.top();
16             while Interval[value][u] = 0
17                 Interval[value][u] ← 1;
18                 value ← LifoLCP.next();
19             Interval[LCP[i]][u] ← 1;
20             B_1[2i][u] ← 1;
21             B_1[2i + 1][u] ← 1;
22             B_2[2i][u] ← 1;
23             B_2[2i + 1][u] ← 1;
24         if i > 0 and LCP[i] > 0 and SA[i − 1] > 0
25             v ← x[SA[i − 1] − 1];
26             Interval[LCP[i]][v] ← 1;
27         B_2[2i] ← Interval[LCP[i]];
28         if LifoLCP.top() ≠ LCP[i]
29             LifoLCP.push(LCP[i]);
```

```
1  Algorithm Bottom-Up-Pass(n, SA, LCP, B1, B2, Σ, σ) [10]
2      Interval[0..max_{i∈[0:n−1]} LCP[i]][0..σ − 1] ← 0;
3      LifoLCP.push(0);
4      foreach i ∈ [0 : n − 1] do
5          proxa ← LCP[i] + 1;
6          proxb ← 1;
7          if i < n − 1 and LCP[i] < LCP[i + 1]
8              while LifoLCP.top() > LCP[i]
9                  proxa ← LifoLCP.pop();
10                 LifoRem.push(proxa);
11             if LifoLCP.top() < LCP[i]
12                 Interval[LCP[i]] ← Interval[proxa];
13         foreach k ∈ Σ : B1[2i][k] = 1 do
14             value ← LifoLCP.top();
15             while Interval[value][k] = 0
16                 Interval[value[k]] ← 1;
17                 value ← LifoLCP.next();
18             Interval[LCP[i]][k] ← 1;
19         B2[2i] ← B2[2i] bit-or Interval[LCP[i]];
20         B2[2i + 1] ← B2[2i + 1] bit-or Interval[LCP[i + 1]];
21         B1[2i + 1] ← B1[2i + 1] bit-or Interval[proxb];
22         proxb ← proxa;
23         B1[2i] ← B1[2i] bit-or Interval[proxa];
24         while LifoRem not empty
25             value ← LifoRem.pop();
26             Interval[value][0..σ − 1] ← 0;
27         if LifoLCP.top() ≠ LCP[i]
28             LifoLCP.push(LCP[i]);
```

2.7 Indexing Weighted Sequences

A string P appears at position i in string S if $P = S[i\,..\,i+|P|-1]$. A *property* $\prod$ of S is a inheritable collection of integer intervals included in $\{1,...,n\}$. We describe every property $\prod$ with an array $\pi[1\,..\,|S|]$ such that the longest interval $I \in \prod$ starting at location i is $\{i,...,\pi[i]\}$. And π can be an arbitrary array contenting $\pi[i] \in \{i-1,...,n\}$ and $\pi[1] \leq \pi[2] \leq ... \leq \pi[n]$. By $Occ_{\pi}(P,S)$, for a string P we define the set of occurrences i of P in S such that $i+|P|-1 \leq \pi[i]$. [12]

PROPERTY INDEXING
Input: A string S of length n over an alphabet Σ and an array π representing a property $\prod$.
Output: For a given pattern string P of length m, compute $\lvert Occ_{\pi}(P,S)\rvert$ or report all elements of $Occ_{\pi}(P,S)$.

Let us take an indexed family $S = (S_j, \pi_j)_{j=1}^{k}$ of strings S_j with properties π_j into consideration. For a string P and an index i, by $Count_S(P,i) = |\{j : i \in Occ_{\pi_j}(P,S_j)\}|$, we define the total number of occurrences of P at the location i in the strings $S_1,...,S_k$ that respect the properties. [12]

A weighted sequence $X = x_1x_2...x_n$ of length $|X| = n$ over an alphabet Σ is a sequence of sets of pairs of the form $X_i = \{(c, p_i^{(X)}(c)) : c \in \Sigma\}$. We denote $p_i^{(X)}(c)$ is the occurrence probability of the letter c at the location $i \in \{1,...,n\}$. For a given i, these values are non-negative and sum up to 1. [12]

The *probability of matching* of a string P at location i of a weighted sequence X equals $Prob_X(P,i) = \prod_{j=1}^{|P|} p_{i+j-1}^{(X)}(P[j])$. If $Prob_X(P,i) \geq \frac{1}{z}$, a string P appears in X at location i. We denote that P is a *solid factor* of X (staring, occurring) at location i. By $Occ_{\frac{1}{z}}(P,X)$, we define the set of all locations where P appears in X. [12]

WEIGHTED INDEXING
Input: A weighted sequence X of length n over an alphabet Σ and a threshold $\frac{1}{z}$.
Output: For a given pattern string p of length m, check if $Occ_{\frac{1}{z}}(P,X) \neq \varnothing$ (*decision* query), compute $\lvert Occ_{\frac{1}{z}}(P,X)\rvert$ (*counting* query), or report all elements of $Occ_{\frac{1}{z}}(P,X)$ (*reporting* query).

Definition 2.7.1 ([12]). We say that an indexed family $S = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ containing strings S_j of length n is a *z-estimation* of a weighted sequence X of length n if and only if, for every string P and position $i \in \{1, ..., n\}$, $Count_S(P,i) = \lfloor Prob_X(P,i)z \rfloor$.

Observation 2.7.1 ([12]). A family $S = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$ is a *z-estimation* of X if and only if for each position i, every string P is a prefix of exactly $t_i(P)$ strings $S_j[i..\pi_j[i]$.

Lemma 2.7.2 ([12]). There exists a unique multiset M_i such that each string P is prefix of exactly $t_i(P)$ strings in M_i.

Definition 2.7.2 ([12]). We say that $P \in M_i$ is *compatible* with $Q \in M_{i+1}$ if $P = \varepsilon$ or $P = cQ^{'}$ for some character $c \in \Sigma$ and a prefix $Q^{'}$ of Q.

Lemma 2.7.3 ([12]). For every $1 \leq i \leq n-1$, there is a one-to-one correspondence from M_{i+1} into M_i such that each $Q \in M_{i+1}$ is matched with a compatible $P \in M_i$.

Theorem 2.7.4 ([12]). Each weighted sequence X has a z-estimation.

We begin with M_{n+1}, which composes of $\lfloor z \rfloor$ copies of ε, and then we iterate over locations $i = n, ..., 1$ transforming M_{i+1} to M_i so that each $P_{j,i+1} \in M_{i+1}$ is exchanged with a compatible string $P_{j,i} \in M_i$. In the meantime, we create the z-estimation $S = (S_j, \pi_j)_{j=1}^{\lfloor z \rfloor}$. In addition, we place $\Pi_j[i]$ to $i + |P_{j,i}| - 1$ and $S_j[i]$ to the leading character of $P_{j,i}$, or an arbitrary character if $P_{j,i} = \varepsilon$. [12]

As well known, a trie is a rooted tree where each node represents a string, and the string corresponding to node u namyly the *label* of u, is defined $L(u)$. The root holds label ε, and the node u' parent with $L(u) = Pc$ for $c \in \Sigma$ is the node v with $L(v) = P$, and c label the edge from P to Pc. The family of solid factors appearing at location i (i.e., strings P such that $t_i(P) > 0$) is closed with respect to prefixes. [12]

We hold M_i utilizing *tokens* in T_i: each $P_{j,i} \in M_i$ is described by a token (with identifier j) posted at the node $u \in T_i$ with $L(u) = P_{j,i}$. For each token j, we hold the node $u \in T_i$ with $L(u) = P_{j,i}$ and the probability $Prob_X(P_{j,i}, i)$. And the number of tokens at the node u is $m_i(L(u))$ and the number of tokens in the subtree rooted at u is $t_i(L(u))$. Now, we define $m_i(u) = m_i(L(u))$ and $t_i(u) = t_i(L(u))$. [12]

Observation 2.7.5 ([12]). The trie T_i contains $\lfloor z \rfloor$ tokens in total and every leaf contains tokens.

Observation 2.7.6 ([12]). If $u \in T_i$ has a non-empty label, $L(u) = cP$, for some $c \in \Sigma$, then T_{i+1} contains a node v with label $L(v) = P$.

For each index i, we transform the solid factor trie T_{i+1} to T_i and turn the tokens so that M_{i+1} is transformed to M_i. Each non-root node $u \in T_i$ has a corresponding node $v \in T_{i+1}$. At times, we apply v as u, otherwise, we make u as a copy of v. We identify a *heavy letter* $h \in \Sigma$ maximising probability $P_i^{(X)}(c)$ over $c \in \Sigma$. we apply v if $L(u)$ begins with h and make a copy of v otherwise. [12]

This method is implemented as follows. First, we make the root of T_i and connect T_{i+1} to the new root utilizing an edge with label h. We define the resulting subtree as $T_{i,h}$, which includes all tokens show in T_{i+1} and may include nodes v with $t_i(v) = 0$. And then, we think all the rest of letters $c \in \Sigma \setminus \{h\}$. For each such letter we will create a subtree $T_{i,c}$ describing solid factors appearing at location i and beginning with character c. At the same time, we create and traverse $T_{i,c}$: we build the children of a node u whereas visiting u for the first time. Meanwhile, at node u with $L(u) = cP$, we keep the probability $Prob_X(cP,i)$ and a pointer to the corresponding node $v \in T_{i,h}$ such that $L(v) = hP$. To build the children of u, we calculate $t_i(cPc^{'})$ for each $c^{'} \in \Sigma$. Furthermore, we make $m_i(cP)$ and set $m_i(cP)$ *token requests* at node v, declaring that $m_i(cP)$ tokens are needed at u. [12]

At last, we turn the tokens and trim the redundant nodes of $T_{i,h}$. We deal the tokens in an arbitrary order. For a token posted at node v of $T_{i,h}$ with $L(v) = hQ$, and that token would describe $Q \in M_{i+1}$. We traverse the line from from v towards the root of T_i holding the probability $Prob_X(L(v^{'}), i)$ at the currently visited node $v^{'}$. First, we review if there is any token request at $v^{'}$. If so, we subscribe with the request, remove it, and finish the traversal. Otherwise, we calculate $m_i(v^{'})$ utilizing the probability. If $v^{'}$ includes less than $m_i(v^{'})$ already dealt tokens, we set our token at $v^{'}$ and end the traversal. Otherwise, we dealt to the $v^{'}$ parent. If $v^{'}$ is a leaf and dose not include any tokens, we remove $v^{'}$ from $T_{i,h}$. If the traversal reaches the T_i's root, we set the token at the root. [12]

Theorem 2.7.7 ([12])**.** For a weighted sequence X of length n over a constant-sized alphabet, one can construct a z-estimation in $\mathcal{O}(nz)$ time.

2.8 Molecular biology

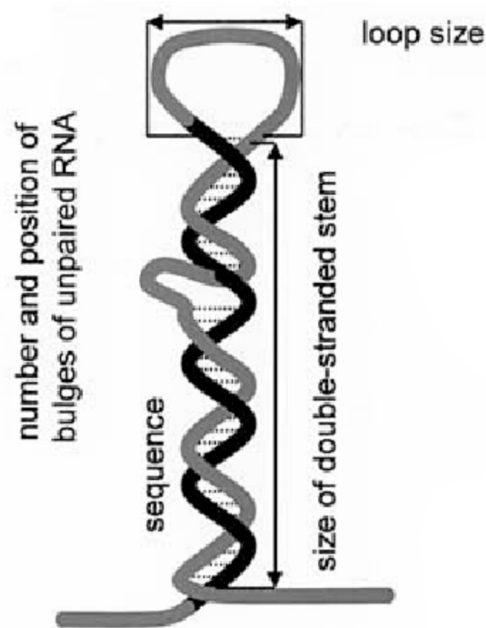
Molecular biology is a branch of biology involving the molecular foundation of biological process between biomolecules in the all kinds of systems of a *cell*. [42] The cell is the essential unit of living organisms. Cells can be divided into two major categories: *prokaryotic cells* and *eukaryotic cells*. Bacterial and archaeal organisms are composed of prokaryotic cells. Protozoa, fungi, plants, and animals are composed of eukaryotic cells. Cells comprise *biomolecules*, which are a general term for molecules naturally present in organisms, including macromolecules such as proteins, carbohydrates, lipids, and nucleic acids, as well as small molecules such as metabolites, secondary metabolites, and natural products.

A nucleoside is a molecule produced from connecting a nucleobase to a deoxyribose ring or ribose, including adenosine (A), thymidine (T), cytidine (C), guanosine (G), uridine (U) and inosine (I). Nucleosides can be phosphorylated by particular kinases in cells to make nucleotides. DNA utilizes deoxynucleotides A, T, C, and G, while RNA utilizes ribonucleotides (with additional hydroxyl (OH) groups on the pentose ring) A, C, G and U.

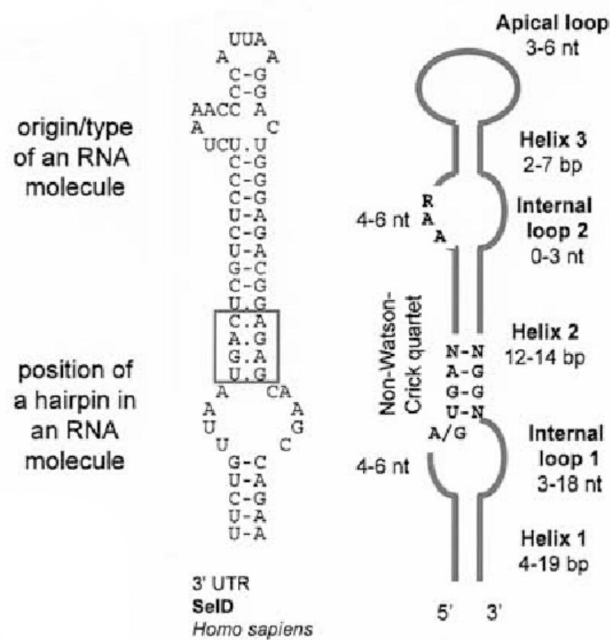
Each DNA molecule is composed of four different nucleotides and may be considered as a sequence of four letters A, C, G, and T representing the four nucleotide bases: adenine, cytosine, guanine and thymine. The Human Genome; for example, is the genetic code - the entire list of three billion letters - required to create a human being. It is quite common, to refer to DNA as the genetic “language” and to A, C, G and T as the “alphabet” of this language. In one sense, a DNA molecule is nothing more

than a sequence formed by concatenating those letters; like $TTGAAGCATA$..., which has a certain biological meaning or function.

We will now briefly describe the structure of *hairpin*, which is the primary motivation of studying maximal palindromes at biological sequence analysis. In RNA or single-stranded DNA, there exists a particular sequence as Stem-loop intramolecular base pairing, called hairpin or hairpin loop. It appears when two regions of the same strand read in opposite directions, frequently complementary in nucleotide sequence, base-pair to form a double helix that ends in an unpaired loop. The finishing structure is an important making module of numerous RNA secondary structures. As an essential secondary structure of RNA, it can guide RNA folding, supply recognition sites for RNA binding proteins, assist structural stability for messenger RNA (mRNA), and act as a substrate for enzymatic reactions. [57] (Fig. 2.5)



(a)



(b)

Fig. 2.5 RNA stem-loops.(a) A schematic overview of an RNA stem-loop depicting the important parameters for the role of such a hairpin RNA. (b) The SECIS stem-loop structure element controlling selenoprotein synthesis. Right: A consensus of a secondary structure of a SECIS element. Left: A specific example of the SECIS element in Homo sapiens. [57]

Chapter 3

Avoided words and Overabundant words

In this chapter, we study the avoided words and overabundant words, and then, we present some efficient algorithms on those words. Moreover, we exhibit some applications of avoided and overabundant words.

This chapter is organised as follows.

In Section 3.1 we introduce the background and contributions of avoided words and overabundant words, that show the motivation and most recent work on avoided words and overabundant words.

In Section 3.2 we present the preliminaries, and give the definition and useful properties of avoided words and overabundant words.

In Section 3.3.1 we provide an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm to compute all ρ-avoided words of length k in a given sequence of length n over a fixed-sized alphabet.

In Section 3.3.2 we provide a time-optimal $\mathcal{O}(\sigma n)$-time algorithm to compute all ρ-avoided words (of any length) in a sequence of length n over an integer alphabet of size σ.

In Section 3.3.3 we provide an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all overabundant words in a sequence x of length n over an integer alphabet.

In Section 3.4 we make available an implementation of our algorithms. Experimental results, using both real and synthetic data, which further highlight the *effectiveness* of this model, show the efficiency and applicability of our implementation in biological sequence analysis.

Finally, in Section 3.5, we give the conclusion of avoided words and overabundant words.

3.1 Background and Contributions

3.1.1 Background

The one-to-one mapping of a DNA molecule to a sequence of letters suggests that DNA analysis can be modelled within the framework of formal language theory [56]. For example, a region within a DNA sequence can be considered as a “word” on a fixed-sized alphabet in which some of its natural aspects can be described by means of certain types of automata or grammars. However, a linguistic analysis of the DNA needs to take into account many distinctive physical and biological characteristics of such sequences: The genome consists of coding regions that encode for polypeptide chains associated with biological functions as well as a plethora of regulatory and potentially

functional non-coding regions, identified through multiple alignment of genomes of several organisms, and termed conserved non-coding elements (CNEs). In addition, it contains large non-coding regions most of which are not linked to any particular function. All these genomic components appear to have many statistical features in common with natural languages [45].

A computational tool oriented towards the systematic search for avoided words is particularly useful for *in silico* genomic research analyses. The search for *absent words* is already undertaken in the recent past and several results exist on the application and computation of such words [1**?** **?** **?**]. However, words which may be present in a genome or in genomic sequences of a specific role (e.g., protein coding segments, regulatory elements, conserved non-coding elements etc) but they are strongly underrepresented—as we can estimate on the basis of the frequency of occurrence of their longest proper factors—may be of particular importance. They can be words of nucleotides which are hardly tolerated because they negatively influence the stability of the chromatin or, more generally, the functional genomic conformation; they can represent targets of restriction endonucleases which may be found in bacterial and viral genomes; or, more generally, they may be short genomic regions whose presence in wide parts of the genome are not tolerated for less known reasons. The understanding of such avoidances is becoming an interesting line of research [17, 55].

On the other hand, short words of nucleotides may be systematically avoided in large genomic regions or whole genomes for entirely different reasons, i.e. just because they play important signaling roles which confine their appearance only in specific positions: consensus sequences for the initiation of gene transcription and of DNA

replication are well-known such oligonucleotides. Other such cases may be insulators, sequences anchoring the chromatin on the nuclear envelope like lamina-associated domains, short sequences like dinucleotide repeat motifs with enhancer activity, and several other cases. Again, we cannot exclude that this area of research could lead to the identification of short sequences of regulatory activities still unknown.

Brendel *et al.* in [19] initiated research into the linguistics of nucleotide sequences that focuses on the concept of words in continuous languages—languages devoid of blanks—and introduced an operational definition of words. The authors suggested a method to measure, for each possible word w of length k, the deviation of its observed frequency from the expected frequency in a given sequence. The values of the deviation, denoted by $dev(w)$, were then used to identify words that are avoided among all possible words of length k. The typical length of avoided (or of overabundant) words of the nucleotide language was found to range from 3 to 5 (tri- to pentamers). The statistical significance of the avoided words was shown to reflect their biological importance. This work, however, was based on the very limited sequence data available at the time: only DNA sequences from two viral and one bacterial genomes were considered. Also note that k might change when considering eukaryotic genomes, the complex dynamics and function of which might impose a more demanding analysis. The authors in [6, 7, 9] studied a similar notion of *unusual words*—based on different definitions than the ones Brendel *et al.* use for expectation and deviation—focusing on the factors of a sequence; based on Brendel *et al.*'s definitions, we focus on any word over the alphabet. More recently, space-efficient detection of unusual words has also been considered [17]; such avoidances is becoming an interesting line of research [55].

Moreover, we extend this study on overabundant words. The motivation comes from molecular biology. Genome dynamics, i.e. the molecular mechanisms generating random mutations in the evolving genome, are quite complex, often presenting self-enhancing features. Thus, it is expected to often give rise to words of nucleotides which will be overabundant, i.e. being present at higher amounts than expected on the basis of their longest proper prefix, longest proper suffix, and longest infix frequencies. One specific such mechanism, which might generate overabundant words, is the following: it is well-known that in a genomic sequence of an initially random composition, the existing relatively long homonucleotide tracts present a higher frequency of further elongation than the frequency expected on the basis of single nucleotide mutations [41]; that is, they present a sort of autocatalytic self-elongation. This feature, in combination with the much higher frequency of transition *vs.* transversion mutation events, generates overabundant words which are homopurinic or homopurimidinic tracts. It is also anticipated that the overabundance of homonucleotide tracts will strongly differentiate between conserved and non-conserved parts of the genome. While this phenomenon is largely free to act within the non-conserved genomic regions, and thus it is expected to generate there large amounts of overabundant words, it is hindered in the conserved genomic regions due to selective constraints.

3.1.2 Contributions

The computational problems can be described as follows. First, given a sequence x of length n, an integer k, and a real number $\rho < 0$, compute the set of ρ-avoided words of length k, i.e. all words w of length k for which $dev(w) \leq \rho$. We call this set the

ρ-*avoided words* of length k in x. Brendel *et al.* did not provide an efficient solution for this computation [19]. Moreover, such a word may be completely absent from x. Consequently, the set of ρ-avoided words can be naïvely computed by considering all possible σ^k words, where σ is the size of the alphabet. Second, given a sequence x of length n and a real number $\rho > 0$, compute the set of ρ-overabundant words, i.e. all words w for which $std(w) \geq \rho$.

Firstly, we present an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all ρ-avoided words of length k in a sequence of length n over a fixed-sized alphabet. For words over an integer alphabet of size σ, the algorithm requires time $\mathcal{O}(\sigma n)$, which is optimal for sufficiently large σ. We also present a time-optimal $\mathcal{O}(\sigma n)$-time algorithm to compute all ρ-avoided words (of any length) in a sequence of length n over an integer alphabet of size σ. We provide a tight asymptotic upper bound for the number of ρ-avoided words over an integer alphabet and the expected length of the longest one. We also prove that the same asymptotic upper bound is tight for the number of ρ-avoided words of fixed length when the alphabet is sufficiently large.

As shown subsequently, the set of absent ρ-avoided words is a subset of the set of minimal absent words of a word. Hence the tight asymptotic bounds for ρ-avoided words are based on the proof we provide for the tightness of the known asymptotic bound on minimal absent words and the tightness of this bound for minimal absent words of fixed length over sufficiently large alphabets.

Secondly, we present an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all ρ-overabundant words (of any length) in a sequence x of length n over an integer alphabet. This result is based on a combinatorial property of the suffix tree $\mathcal{T}$ of x

that we prove here: the number of distinct factors of x whose longest infix is the label of an explicit node of $\mathcal{T}$ is no more than $3n-4$. We further show that the presented algorithm is time-optimal by proving that $\mathcal{O}(n)$ is a tight upper bound for the number of ρ-overabundant words. We further show that the presented algorithm is time-optimal by proving that $\mathcal{O}(n)$ is a tight upper bound for the number of ρ-overabundant words. Finally, we pose an open question of combinatorial nature on the maximum number of overabundant words that a sequence of length n over an alphabet of size $\sigma > 1$ can contain.

Analogously to avoided words [3, 19, 31], many different models and algorithms exist for identifying words that are in abundance in a given sequence; see for instance [20, 24]. In this chapter, we make use of the biologically justified model by proving non-trivial combinatorial properties, we show that it admits *efficient* computation for overabundant words as well.

We make available an open-source implementation of our algorithm. Experimental results, using both real and synthetic data, which further highlight the *effectiveness* of this model, show its efficiency and applicability. Specifically, using our method we confirm that restriction endonucleases which target self-complementary sites are not found in eukaryotic sequences [55]. In addition, we apply our algorithm in the case of CNEs, which are classes of sequences whose functions in our genomes remain largely enigmatic [34, 51]. We observe interesting patterns of occurring avoided words within CNEs compared to CNE-like sequences (surrogates) that are in accordance with their distinct sequence characteristics which classify them from other non-functional sequences [50, 52].

3.2 Preliminaries

3.2.1 Definitions and Notations

We begin with basic definitions and notation generally following [22]. We denote the *reverse* word of x by $\mathsf{rev}(x)$, i.e. $\mathsf{rev}(x) = x[n-1]x[n-2]\ldots x[1]x[0]$. We say that x is *a power* of a word y if there exists a positive integer k, $k > 1$, such that x is expressed as k consecutive concatenations of y; we denote that by $x = y^k$. Let $f(w)$ denote the *observed frequency*, that is, the number of occurrences of a non-empty word w in word x. If $f(w) = 0$ for some word w, then w is called *absent* (which is denoted by $w \not\preceq x$), otherwise, w is called *occurring*.

By $f(w_p)$, $f(w_s)$, and $f(w_i)$ we denote the observed frequency of the longest proper prefix w_p, suffix w_s, and infix w_i of w in x, respectively. We can now define the *expected frequency* of word w, $|w| > 2$, in x as in Brendel et al. [19]:

$$E(w) = \frac{f(w_p) \times f(w_s)}{f(w_i)}, \text{ if } f(w_i) > 0; \text{ else } E(w) = 0. \tag{3.1}$$

The above definition can be explained intuitively as follows. Suppose we are given $f(w_p)$, $f(w_s)$, and $f(w_i)$. Given an occurrence of w_i in x, the probability of it being preceded by $w[0]$ is $\frac{f(w_p)}{f(w_i)}$ as $w[0]$ precedes exactly $f(w_p)$ of the $f(w_i)$ occurrences of w_i. Similarly, this occurrence of w_i is also an occurrence of w_s with probability $\frac{f(w_s)}{f(w_i)}$. Although these two events are not always independent, the product $\frac{f(w_p)}{f(w_i)} \times \frac{f(w_s)}{f(w_i)}$ gives a good approximation of the probability that an occurrence of w_i at position j implies an occurrence of w at position $j-1$. It can be seen then that by multiplying this product by

the number of occurrences of w_i we get the above formula for the expected frequency of w.

Moreover, to measure the deviation of the observed frequency of a word w from its expected frequency in x, we define the *deviation* (χ^2 test) of w as:

$$dev(w) = \frac{f(w) - E(w)}{\max\{\sqrt{E(w)}, 1\}}. \quad (3.2)$$

For more details on the *biological* justification of these definitions see [19].

Using the above definitions and two given thresholds, we can classify a word w as either *avoided*, *common*, or *overabundant* in x. In particular, for two given thresholds $\rho_1 < 0$ and $\rho_2 > 0$, a word w is called ρ_1-*avoided* if $std(w) \leq \rho_1$, ρ_2-*overabundant* if $std(w) \geq \rho_2$, and (ρ_1, ρ_2)-*common* otherwise. In this chapter, we consider the following computational problems.

AVOIDEDWORDSCOMPUTATION

Input: A word x of length n, an integer $k > 2$, and a real number $\rho < 0$

Output: All ρ-avoided words of length k in x

ALLAVOIDEDWORDSCOMPUTATION

Input: A word x of length n and a real number $\rho < 0$

Output: All ρ-avoided words in x

ALLOVERABUNDANTWORDSCOMPUTATION
Input: A word x of length n and a real number $\rho > 0$
Output: All ρ-overabundant words in x

3.2.2 Tight Asymptotic Bounds on Minimal Absent Words

In this section, we provide a tight asymptotic upper bound for the number of ρ-avoided words over an integer alphabet and the expected length of the longest one. We also prove that the same asymptotic upper bound is tight for the number of ρ-avoided words of fixed length when the alphabet is sufficiently large.

Definition 3.2.1 ([?])**.** An absent word w of x is *minimal* if and only if all proper factors of w occur in x.

We first show that the known asymptotic upper bound on the number of minimal absent words of a word is tight.

Lemma 3.2.1 ([21])**.** The asymptotic upper bound $\mathcal{O}(\sigma n)$ on the number of minimal absent words of a word of length n over an alphabet of size σ is tight if $2 \leq \sigma \leq n$.

Proof. To prove that the bound is tight it suffices to construct a word with these many minimal absent words asymptotically.

Let $\Sigma = \{a_1, a_2\}$, i.e. $\sigma = 2$, and consider the word $x = a_2 a_1^{n-2} a_2$ of length n. All words of the form $a_2 a_1^k a_2$ for $0 \leq k \leq n-3$ are minimal absent words in x. Hence x has at least $n-2 = \Omega(n)$ minimal absent words.

Let $\Sigma = \{a_1, a_2, a_3, \ldots, a_\sigma\}$ with $3 \leq \sigma \leq n$ and consider the word $x = a_2 a_1^k a_3 a_1^k a_4 a_1^k \ldots a_i a_1^k a_{i+1} \ldots a_\sigma a_1^k a_1^m$, where $k = \lfloor \frac{n}{\sigma-1} \rfloor - 1$ and $m = n - (\sigma-1)(k+1)$. Note that x is of length n. Further note that $a_i a_1^j$ is a factor of x, for all $2 \leq i \leq \sigma$ and $0 \leq j \leq k$. Similarly, $a_1^j a_l$ is a factor of x, for all $3 \leq l \leq \sigma$ and $0 \leq j \leq k$. Thus all proper factors of all the words in the set $S = \{a_i a_1^j a_l \mid 0 \leq j \leq k,\ 2 \leq i \leq \sigma,\ 3 \leq l \leq \sigma\}$ occur in x. However, the only words in S that occur in x are the ones of the form $a_i a_1^k a_{i+1}$, for $2 \leq i < \sigma$. Hence x has at least $(\sigma-1)(\sigma-2)(k+1) - (\sigma-2) = (\sigma-1)(\sigma-2)\lfloor \frac{n}{\sigma-1} \rfloor - (\sigma-2) = \Omega(\sigma n)$ minimal absent words. □

In the following lemma we show that, for sufficiently large alphabets, $\mathcal{O}(\sigma n)$ is a tight asymptotic bound for the number of minimal absent words of fixed length.

Lemma 3.2.2. The asymptotic upper bound $\mathcal{O}(\sigma n)$ on the number of minimal absent words of fixed length of a word of length n over an alphabet of size σ is tight if $\sqrt{n} + 1 \leq \sigma \leq n$.

Proof. Let $\Sigma = \{a_1, a_2, a_3, \ldots, a_\sigma\}$ be an alphabet of size σ. We will show that we can construct words of any length n, with $\sigma \leq n \leq \sigma(\sigma-1)$, that have $\Omega(\sigma n)$ minimal absent words of length 3.

We first construct the strings (blocks) $B_i = a_{i+1} a_i a_{i+2} a_i \ldots a_{i+j} a_i \ldots a_\sigma a_i$, for $1 \leq i \leq \sigma - 1$. Note that $|B_i| = 2(\sigma - i)$ and that a letter a_i occurs in B_j if and only

if $j \leq i$. We then consider the word $x = B_1 B_2 \ldots B_i \ldots B_{\sigma-1}$ which has length $|x| = \sum_{i=1}^{\sigma-1} 2(\sigma - i) = \sigma(\sigma - 1)$.

Now consider any prefix y of x with $|y| > 2(\sigma - 1)$. Then $y = B_1 B_2 ... B_{j-1} \overline{B_j}$, where $\overline{B_j}$ is a prefix of B_j for some $j > 1$. For any $i < j$ the words of length 3 with a_i as the mid-letter that occur in y are the ones in the set $U_i = \{a_\ell a_i a_\ell \mid 1 \leq \ell \leq i-2\} \cup \{a_k a_i a_{k+1} \mid i+1 \leq k \leq \sigma - 1\} \cup \{a_{i-2} a_i a_{i-1}\} \cup \{a_\sigma a_i a_{i+2}\}$, with the last singleton not included if $i = j-1$ and $\overline{B_j} = \varepsilon$. We thus have $|U_i| \leq \sigma$.

We notice that the strings of the form $a_k a_i$ for all $k \in P_i = \{1, 2, \ldots, \sigma\} \setminus \{i-1, i\}$ occur in y and similarly the strings of the form $a_i a_\ell$ for all $\ell \in S_i = \{1, 2, \ldots, \sigma\} \setminus \{i, i+1\}$ occur in y. Hence, all proper factors of all strings in $V_i = \{a_k a_i a_\ell \mid k \in P_i, \ell \in S_i\}$ occur in y and $|V_i| = (\sigma - 2)^2$. Then all the words in $M_i = V_i \setminus U_i$ are minimal absent words of y of length 3 with mid-letter a_i and they are at least $(\sigma - 2)^2 - \sigma$. Now, since $|B_i| < 2\sigma$ for all i, we have that $j > \frac{|y|}{2\sigma}$. Hence $\sum_{i=1}^{j-1} |M_i| \geq ((\sigma - 2)^2 - \sigma) \times \frac{|y|}{2\sigma}$. Since the sets M_i are pairwise disjoint it then follows that y has $\Omega(\sigma|y|)$ minimal absent words of length 3.

Hence, given an alphabet of size σ we can construct words of any length n, such that $2\sigma < n \leq \sigma(\sigma - 1)$, that have $\Omega(\sigma n)$ minimal absent words of length 3.

Note that when $\sigma \leq n \leq 2\sigma$ the example of $y = a_1 a_2 a_3 \ldots a_\sigma$ (possibly padded with a_σ's) gives the desired result as at most σ out of the σ^2 possible combinations $a_i a_j$ (of length 2) occur in y, while all proper factors of all such combinations occur in y. □

3.2.3 Useful Properties of Avoided Words

In this section, we provide some useful insights of combinatorial nature which were not considered by Brendel *et al*. [19]. By the definition of ρ-avoided words it follows that a word w may be ρ-avoided even if it is absent from x. In other words, $dev(w) \leq \rho$ may hold for either $f(w) > 0$ (occurring) or $f(w) = 0$ (absent).

Example 3.2.1. Consider again the word $x = \texttt{AGCGCGACGTCTGTGT}$, $k = 3$, and $\rho = -0.4$.

- word $w_1 = \texttt{CGT}$, at position 7 of x, is an *occurring* ρ-avoided word:

$$E(w_1) = 3 \times 3/6 = 1.5,\ dev(w_1) = (1 - 1.5)/\sqrt{1.5} = -0.408248.$$

- word $w_2 = \texttt{AGT}$ is an *absent* ρ-avoided word:

$$E(w_2) = 1 \times 3/6 = 0.5,\ dev(w_2) = (0 - 0.5)/1 = -0.5.$$

This means that a naïve computation should consider *all* possible σ^k words. Then for each possible word w, the value of $dev(w)$ can be computed via pattern matching on the suffix tree of x. In particular, we can search for the occurrences of w, w_p, w_s, and w_i in x in time $\mathcal{O}(k)$ [22]. In order to avoid this inefficient computation, we exploit the following crucial lemmas.

Lemma 3.2.3. Any absent ρ-avoided word w in x is a minimal absent word of x.

Proof. For w to be a ρ-avoided word it must hold that

$$dev(w) = \frac{f(w) - E(w)}{\max\{\sqrt{E(w)}, 1\}} \leq \rho < 0.$$

This implies that $f(w) - E(w) < 0$, which in turn implies that $E(w) > 0$ since $f(w) = 0$. From $E(w) = \frac{f(w_p) \times f(w_s)}{f(w_i)} > 0$, we conclude that $f(w_p) > 0$ and $f(w_s) > 0$ must hold. Since $f(w) = 0$, $f(w_p) > 0$, and $f(w_s) > 0$, w is a minimal absent word of x: all proper factors of w occur in x. □

Lemma 3.2.4. Let w be a word occurring in x and $\mathcal{T}(x)$ be the suffix tree of x. Then, if w_p is a path-label of an implicit node of $\mathcal{T}(x)$, $dev(w) \geq 0$.

Proof. According $f(w_p)$, $f(w_s)$, and $f(w_i)$ denote the observed frequency of the longest proper prefix w_p, suffix w_s, and infix w_i of w in x, respectively. For any w that occurs in x it holds that $f(w_i) \geq f(w_s)$, which implies that $f(w_p) \geq \frac{f(w_p) \times f(w_s)}{f(w_i)} = E(w)$. Furthermore, by the definition of the suffix tree, if w occurs in x and w_p is a path-label of an implicit node then $f(w_p) = f(w)$. It thus follows that $f(w) - E(w) = f(w_p) - E(w) \geq 0$, and since $\max\{1, \sqrt{E(w)}\} > 0$, the claim holds. □

Lemma 3.2.5. The number of ρ-avoided words of length $k > 2$ in a word of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$; in particular, this number is no more than $(\sigma + 1)n - k + 1$. The asymptotic upper bound $\mathcal{O}(\sigma n)$ is tight if $\sqrt{n} + 1 \leq \sigma \leq n$.

Proof. By Lemma 3.2.3, every ρ-avoided word is either occurring or a minimal absent word. It is known that the number of minimal absent words in a word of length n is smaller than or equal to σn [47]. Clearly, the occurring ρ-avoided words in a word of length n are at most $n-k+1$. Therefore the number of ρ-avoided words of length k are no more than $(\sigma+1)n-k+1$. This implies that $\mathcal{O}(\sigma n)$ is an asymptotic upper bound. In the case of an alphabet of size $\sqrt{n}+1 \leq \sigma \leq n$, it follows from Lemma 3.2.2 that there exist words with $\Omega(\sigma n)$ minimal absent words of a fixed length $k > 2$. Consider such a word x, the respective k, and some $\rho \geq -\frac{1}{n}$. Let w be any minimal absent word of x. We have that $f(w_p) \geq 1$, $f(w_s) \geq 1$, and $f(w_i) \leq n$; and hence $E(w) \geq \frac{1}{n}$. Since $f(w) = 0$, it follows that $dev(w) \leq -\frac{1}{n} \leq \rho$. Thus, every minimal absent word of x is ρ-avoided, and since there are $\Omega(\sigma n)$ of them of length k, we conclude that $\mathcal{O}(\sigma n)$ is a tight asymptotic bound in this case. □

3.2.4 Useful Properties of Overabundant Words

In this section, we prove some properties that are useful for designing the time-optimal algorithm presented in the next section.

Fact 3.2.6. Given a word x of length n over an alphabet of size σ, the number of words w for which $dev(w)$ is defined is $\mathcal{O}((\sigma n)^2)$.

Proof. For a word w over Σ, $dev(w)$ is only defined if $w_i \preceq x$. Hence the words w for which $dev(w)$ is defined are of the form aub for some non-empty $u \preceq x$ and $a, b \in \Sigma$.

For each distinct factor $u \neq \varepsilon$ of x there are σ^2 words of the form aub, $a, b \in \Sigma$. Since there are $\mathcal{O}(n^2)$ distinct factors in a word of length n, the fact follows. □

Fact 3.2.7. Every word w that does not occur in x and for which $dev(w)$ is defined has $dev(w) \leq 0$.

Proof. For such a word we have that $E(w) \geq 0$ and that $f(w) = 0$ and hence $dev(w) = \frac{f(w)-E(w)}{\max\{\sqrt{E(w)},1\}} \leq 0$. □

Naïve algorithm. By using Fact 3.2.7, we can compute $dev(w)$, for each factor w of x, thus solving Problem ALLOVERABUNDANTWORDSCOMPUTATION. There are $\mathcal{O}(n^2)$ such factors, however, which make this computation inefficient. The overall time complexity of this naïve algorithm can be more than $\mathcal{O}(n^3)$.

Fact 3.2.8. Given a factor w of a word x, if w_i corresponds to an implicit node in the suffix tree $\mathcal{T}(x)$, then so does w_p.

Proof. A factor w' of x corresponds to an implicit node $\mathcal{T}(x)$ if and only if every occurrence of it in x is followed by the same unique letter $b \in \Sigma$. Hence, since $w_p = aw_i$ for some $a \in \Sigma$, if w_i is always followed by, say, $b \in \Sigma$, every occurrence of w_p in x must also always be followed by b. Thus w_p corresponds to an implicit node as well. □

Lemma 3.2.9. If w is a factor of a word x and w_i corresponds to an implicit node in $\mathcal{T}(x)$, then $dev(w) = 0$.

Proof. If a word $w' \preceq x$ corresponds to an implicit node along the edge (u,v) in $\mathcal{T}(x)$ and $\mathcal{L}(v) = w$ then the number of occurrences of w' in x is equal to that of w.

If w_i corresponds to an implicit node on edge (u,v) it follows immediately that $f(w_i) = f(w_s)$, as either w_s also corresponds to an implicit node in the same edge or $w_s = \mathcal{L}(v)$. In addition, from Fact 3.2.8 we have that w_p is an implicit node as well and it similarly follows that $f(w_p) = f(w)$. We thus have $E(w) = \frac{f(w_p) \times f(w_s)}{f(w_i)} = f(w)$ and hence $dev(w) = \frac{f(w) - E(w)}{\max\{\sqrt{E(w)}, 1\}} = 0$. □

Based on these properties, the aim of the algorithm in the next section is to find the factors of x whose longest infix corresponds to an explicit node and check if they are ρ-overabundant. More specifically, for each explicit node v in $\mathcal{T}(x)$, such that $\mathcal{L}(v) = y$, we aim at identifying the factors of x that have y as their longest infix (i.e. factors of the form ayb, $a, b \in \Sigma$). We will do that by identifying the factors of x that have y as their longest proper suffix (i.e. factors of the form ay, $a \in \Sigma$) and then checking for each of these the different letters that succeed it in x. Then we can check in time $\mathcal{O}(1)$ if each of these words is ρ-overabundant.

Note that the algorithm presented in Section 3.3.3 is fundamentally different and in a sense more involved than the one presented in [3] for the computation of *occurring* ρ-avoided words (note that a ρ-avoided word can be *absent*). This is due to the fact that for occurring ρ-avoided words we have the stronger property that w_p must correspond to an explicit node.

Theorem 3.2.10. Given a word x of length n, the number of distinct factors of x of the form ayb, where $a,b \in \Sigma$ and $y \neq \varepsilon$ is the label of an explicit node of $\mathcal{T}(x)$, is no more than $3n-2-2\sigma_x$.

Proof. Let S be the set of all explicit or implicit nodes in $\mathcal{T}(x)$ of the form yb such that y is represented by an explicit node other than the root. We have at most $2n-2-\sigma_x$ of them; there are at most $2n-2$ edges in $\mathcal{T}(x)$, but σ_x of them are outgoing from the root. For such a word yb, the number of factors of x of the form ayb is equal to the degree of the node representing $\mathsf{rev}(yb)$ in $\mathcal{T}(\mathsf{rev}(x))$.

For every node in S, we obtain a distinct node in $\mathcal{T}(\mathsf{rev}(x))$. Let us suppose that k_1 of these nodes are non-root internal explicit nodes, k_2 are leaves, and the rest $2n-2-\sigma_x-k_1-k_2$ are implicit nodes. Each internal explicit node u contributes at most $deg(u)$ factors, where $deg(u)$ is the number of outgoing edges of node u, each leaf contributes 0 factors, and each implicit node contributes at most 1 factor.

Hence the number of such factors would be maximised if we obtained all the non-root internal explicit nodes and no leaves in $\mathcal{T}(\mathsf{rev}(x))$. Let $\mathcal{T}(\mathsf{rev}(x))$ have m non-root internal explicit nodes. The resulting upper bound then is $\sum_{u\in\mathcal{T}(\mathsf{rev}(x))\setminus\{root\}} deg(u) + (2n-2-\sigma_x-m) \leq n+m-\sigma_x+(2n-2-\sigma_x-m) = 3n-2-2\sigma_x$.

Note that $\sum_{u\in\mathcal{T}(\mathsf{rev}(x))\setminus\{root\}} deg(u) \leq n+m-\sigma_x$ since there are at most n edges from explicit internal nodes to leaves and m edges to other internal nodes; σ_x of these are outgoing from the root. □

Corollary 3.2.1. The number of ρ-overabundant words in a word x of length n is at most $3n-2-2\sigma_x$.

Proof. By Fact 3.2.7, Lemma 3.2.9, and symmetry, it follows that the ρ-overabundant words in x are factors of x of the form ayb, where $a, b \in \Sigma$, such that $y \neq \varepsilon$ is represented by an explicit node in $\mathcal{T}(x)$ and $\mathsf{rev}(y)$ represented by an explicit node in $\mathcal{T}(\mathsf{rev}(x))$. Hence they are a subset of the set of words considered in Theorem 3.2.10. □

Lemma 3.2.11. The number of ρ-overabundant words in a word x of length n over a binary alphabet (e.g. $\Sigma = \{\mathtt{a}, \mathtt{b}\}$) is no more than $2n-4$.

Proof. For every internal explicit node u of $\mathcal{T}(x)$, other than the root, let $deg'(u)$ be $deg(u)+1$ if node u is terminal and $deg(u)$ otherwise. The sum of $deg'(u)$ over the internal explicit non-root nodes of $\mathcal{T}(x)$ is no more than $2n-4$ (ignoring the case when $x = \alpha^n, \alpha \in \Sigma$). We will show that, for each such node, the number of ρ-overabundant words with $w_i = \mathcal{L}(u)$ as their longest proper infix is at most $deg'(u)$.

- *Case I: $deg'(u) = 2$.*
  - *Subcase 1: $deg(u) = 1$.* Node u is terminal and it has an edge with label α. We can then have at most 2 ρ-overabundant words with w_i as their longest proper infix: $\mathtt{a}w_i\alpha$ and $\mathtt{b}w_i\alpha$.
  - *Subcase 2: $deg(u) = 2$.* Node u is not terminal and it has an edge with label $\mathtt{a}$ and an edge with label $\mathtt{b}$. If only one of $\mathtt{a}w_i$ and $\mathtt{b}w_i$ occurs in x we are done. If both of them occur in x we argue as follows (irrespective of whether w_i is also a prefix of x):

    If $\mathtt{a}w_i\mathtt{a}$ is ρ-overabundant, then

    $f(\mathtt{a}w_i\mathtt{a}) - f(\mathtt{a}w_i) \times f(w_i\mathtt{a})/f(w_i) \geq \rho > 0 \Rightarrow$

$f(\mathtt{a}w_i\mathtt{a})/f(\mathtt{a}w_i) > f(w_i\mathtt{a})/f(w_i) \Leftrightarrow$

$1 - f(\mathtt{a}w_i\mathtt{a})/f(\mathtt{a}w_i) < 1 - f(w_i\mathtt{a})/f(w_i) \Leftrightarrow$

$f(\mathtt{a}w_i\mathtt{b})/f(\mathtt{a}w_i) < f(w_i\mathtt{b})/f(w_i) \Leftrightarrow$

$f(\mathtt{a}w_i\mathtt{b}) - f(\mathtt{a}w_i) \times f(w_i\mathtt{b})/f(w_i) < 0$

and hence $\mathtt{a}w_i\mathtt{b}$ is not ρ-overabundant.

(Similarly for $\mathtt{b}w_i\mathtt{a}$ and $\mathtt{b}w_i\mathtt{b}$.)

- *Case II: $deg'(u) = 3$.* Node u is terminal and it has an edge with label $\mathtt{a}$ and an edge with label $\mathtt{b}$. If only one of $\mathtt{a}w_i$ and $\mathtt{b}w_i$ occurs in x or if both of them occur in x, but w_i is not a prefix of x, we can have at most 2 ρ-overabundant words with w_i as the proper longest infix; this can be seen by looking at the node representing $\mathsf{rev}(w_i)$ in $\mathcal{T}(\mathsf{rev}(x))$, which falls in *Case I*.

  So we only have to consider the case where both $\mathtt{a}w_i$ and $\mathtt{b}w_i$ occur in x and w_i is a prefix of x. For this case, we assume without loss of generality that $\mathtt{a}w_i$ is a suffix of x. If $\mathtt{a}w_i\mathtt{a}$ is ρ-overabundant, then

  $f(\mathtt{a}w_i\mathtt{a}) - f(\mathtt{a}w_i) \times f(w_i\mathtt{a})/f(w_i) \geq \rho > 0 \Rightarrow$

  $f(\mathtt{a}w_i\mathtt{a})/f(\mathtt{a}w_i) > f(w_i\mathtt{a})/f(w_i) \Leftrightarrow$

  $1 - f(\mathtt{a}w_i\mathtt{a})/f(\mathtt{a}w_i) < 1 - f(w_i\mathtt{a})/f(w_i) \Leftrightarrow$

  $(f(\mathtt{a}w_i\mathtt{b}) + 1)/f(\mathtt{a}w_i) < (f(w_i\mathtt{b}) + 1)/f(w_i) \Rightarrow$

  $f(\mathtt{a}w_i\mathtt{b})/f(\mathtt{a}w_i) < (f(w_i\mathtt{b})/f(w_i) \Leftrightarrow$

  $f(\mathtt{a}w_i\mathtt{b}) - f(\mathtt{a}w_i) \times f(w_i\mathtt{b})/f(w_i) < 0$

  and hence $\mathtt{a}w_i\mathtt{b}$ is not ρ-overabundant.

  Thus in this case we can have at most $3 = deg'(u)$ ρ-overabundant words.

We can thus have at most $deg'(u)$ ρ-overabundant words for each internal explicit non-root node of $\mathcal{T}(x)$. This concludes the proof. □

Lemma 3.2.12. The number of ρ-overabundant words in a word of length n is $\mathcal{O}(n)$ and this asymptotic bound is tight. There exists a word over the binary alphabet with $2n-6$ ρ-overabundant words.

Proof. The asymptotic bound follows directly from Corollary 3.2.1. The tightness of the asymptotic bound can be seen by considering word $x = ba^{n-2}b$, $a,b \in \Sigma$, of length n and some ρ such that $0 < \rho < 1/n$. Then for every prefix w of x of the form ba^k and for every suffix w' of x of the form a^kb, $2 \leq k \leq n-2$, we have that $f(w_p) = f(w'_s) = 1$, $f(w_s) = f(w'_p) = n-k-1$, and $f(w_i) = f(w'_i) = n-k$. Hence for any w we have $std(w) = 1 - \frac{1\times(n-k-1)}{n-k} = \frac{1}{n-k} > \rho$. For instance, for $w = ba^{n-2}$, we have $std(w) = 1/2$. There are $2n-6 = \Omega(n)$ such factors and hence at least these many ρ-overabundant words. □

Corollary 3.2.2. The number of (ρ_1,ρ_2)-common words in a word of length n over an alphabet of size σ is $\mathcal{O}((\sigma n)^2)$.

Proof. By Fact 3.2.6 we know that $std(w)$ is defined for $\mathcal{O}((\sigma n)^2)$ words. The ρ_1-avoided ones are $\mathcal{O}(\sigma n)$ [3], while the ρ_2-overabundant are $\mathcal{O}(n)$ by Corollary 3.2.1. Hence the number of (ρ_1,ρ_2)-common words is $\mathcal{O}((\sigma n)^2)$. □

3.3 Algorithms

3.3.1 Computation of Avoided words

In this section, we present Algorithm AVOIDEDWORDS for computing all ρ-avoided words of length k in a given word x. The algorithm builds the suffix tree $\mathcal{T}(x)$ for word x, and then prepares $\mathcal{T}(x)$ to allow constant-time observed frequency queries. This is mainly achieved by counting the terminal nodes in the subtree rooted at node v for every node v of $\mathcal{T}(x)$. Additionally during this pre-processing, the algorithm computes the word-depth of v for every node v of $\mathcal{T}(x)$. By Lemma 3.2.3, ρ-avoided words are classified as either occurring or (minimal) absent, therefore Algorithm AVOIDEDWORDS calls Routines ABSENTAVOIDEDWORDS and OCCURRINGAVOIDEDWORDS to compute both classes of ρ-avoided words in x. The outline of Algorithm AVOIDEDWORDS is as follows.

AVOIDEDWORDS(x,k,ρ)

1 $\mathcal{T}(x) \leftarrow$ SUFFIXTREE(x)
2 **for** each node $v \in \mathcal{T}(x)$ **do**
3 $\quad\quad \mathcal{D}(v) \leftarrow$ word-depth of v
4 $\quad\quad \mathcal{C}(v) \leftarrow$ number of terminal nodes in the subtree rooted at v
5 ABSENTAVOIDEDWORDS(x,k,ρ)
6 OCCURRINGAVOIDEDWORDS(x,k,ρ)

Computing Absent Avoided Words

In Lemma 3.2.3, we showed that each absent ρ-avoided word is a minimal absent word. Thus, Routine ABSENTAVOIDEDWORDS starts by computing all minimal absent words in x; this can be done in time and space $\mathcal{O}(n)$ for a fixed-sized alphabet or in time $\mathcal{O}(\sigma n)$ for integer alphabets [**? ?**]. Let $< (i,j), \alpha >$ be a tuple representing a minimal absent word in x, where for some minimal absent word w of length $|w| > 2$, $w = x[i\,..\,j]\alpha$, $\alpha \in \Sigma$; this representation is clearly unique.

Intuitively, the idea is to check the length of every minimal absent word. If a tuple $< (i,j), \alpha >$ represents a minimal absent word w of length $k = j - i + 2$, then the value of $dev(w)$ is computed to determine whether w is an absent ρ-avoided word. Note that, if $w = x[i\,..\,j]\alpha$ is a minimal absent word, then $w_p = x[i\,..\,j]$, $w_i = x[i+1\,..\,j]$, and $w_s = x[i+1\,..\,j]\alpha$ occur in x by Definition 3.2.1. Thus, there are three (implicit or explicit) nodes in $\mathcal{T}(x)$ path-labelled w_p, w_i, and w_s, respectively.

The observed frequencies of w_p, w_i, and w_s are already computed during the pre-processing of $\mathcal{T}(x)$. For an explicit node v of $\mathcal{T}(x)$, path-labelled $w' = x[i'\,..\,j']$, the value $\mathcal{C}(v)$, which is the number of terminal nodes in the subtree rooted at v, is equal to the number of occurrences (observed frequency) of w' in x. For an implicit node along the edge (u,v) path-labelled w'', the number of occurrences of w'' is equal to $\mathcal{C}(v)$ (and not $\mathcal{C}(u)$). The implementation of this procedure is given in Routine ABSENTAVOIDEDWORDS.

AbsentAvoidedWords(x, k, ρ)

```
1   A ← MinimalAbsentWords(x)
2   for each tuple < (i, j), α >∈ A such that k = j − i + 2 do
3       u_p ← Node(i, j)
4       if IsImplicit(u_p) then
5           (u, v) ← Edge(u_p)
6           f_p ← C(v)
7       else  f_p ← C(u_p)
8       u_i ← Node(i + 1, j)
9       if IsImplicit(u_i) then
10          (u, v) ← Edge(u_i)
11          f_i ← f_s ← C(v)
12      else  f_i ← C(u_i)
13          u_s ← Child(u_i, α)
14          f_s ← C(u_s)
15      E ← f_p × f_s / f_i
16      if (0 − E)/(max{1, √E}) ≤ ρ then
17          Report(x[i .. j]α)
```

Computing Occurring Avoided Words

For a ρ-avoided word to be an occurring one, it has to occur at least once in x. Thus, the idea of Routine OCCURRINGAVOIDEDWORDS is to traverse $\mathcal{T}(x)$. For each node v such that the word-depth of v, $\mathcal{D}(v) = k$, and its path-label, $\mathcal{L}(v) = w$, the procedure computes $dev(w)$ and determines whether w is an occurring ρ-avoided word accordingly. The routine computes the expected frequencies of w, w_p, w_i, and w_s in a similar way as in Routine ABSENTAVOIDEDWORDS.

The main difference is that once node v path-labelled w, such that $|w| = k$, is reached during the traversal of $\mathcal{T}(x)$, then its parent node v_p path-labelled w_p, such that $|w_p| = k-1$, is considered. Then, the suffix-link of v_p is followed to node v_i path-labelled w_i, such that $|w_i| = k-2$. Finally, the child v_s of v_i, such that the label of edge (v_i, v_s) starts with the same letter as edge (v_p, v) is considered; here the path-label of v_s is w_s and $|w_s| = k-1$. The implementation of this procedure is given in Routine OCCURRINGAVOIDEDWORDS.

Lemma 3.2.4 suggests that for each occurring ρ-avoided word w, w_p is a path-label of an explicit node v of $\mathcal{T}(x)$. Thus, for each internal node v such that $\mathcal{D}(v) = k-1$ and $\mathcal{L}(v) = w_p$, Routine OCCURRINGAVOIDEDWORDS computes $dev(w)$, where $w = w_p\alpha$, $\alpha \in \Sigma$, is a path-label of a child (explicit or implicit) node of v. Note that if w_p is a path-label of an explicit node v then w_i is a path-label of an explicit node u of $\mathcal{T}(x)$; node u is well-defined and it is the node pointed at by the suffix-link of v. The implementation of this procedure is given in Routine OCCURRINGAVOIDEDWORDS.

```
OccurringAvoidedWords(x, k, ρ)

 1  N ← an empty stack
 2  Push(N, root(T(x)))
 3  while N is not empty do
 4      u ← Pop(N)
 5      for each edge (u, v) of T(x) do
 6          if D(v) < k − 1 then
 7              Push(N, v)
 8          elseif D(v) = k − 1 then
 9              f_p ← C(v)
10              f_i ← C(suffix-link[v])
11              for each v′ = Child(v, α), α ∈ Σ do
12                  f_w ← C(v′)
13                  f_s ← C(Child(suffix-link[v], α))
14                  E ← f_p × f_s / f_i
15                  if (f_w − E)/(max{1, √E}) ≤ ρ then
16                      Report(L(v′)[0..k − 1])
```

Analysis of the Algorithm

Lemma 3.3.1. Given a word x, an integer $k > 2$, and a real number $\rho < 0$, Algorithm AVOIDEDWORDS computes all ρ-avoided words of length k in x.

Proof. By definition, a ρ-avoided word w is either an absent ρ-avoided word or an occurring one. Hence, the proof of correctness relies on Lemma 3.2.3 and Lemma 3.2.4.

First, Lemma 3.2.3 indicates that an absent ρ-avoided word in x is necessarily a minimal absent word. Routine ABSENTAVOIDEDWORDS considers each minimal absent word w and verifies if w is a ρ-avoided word of length k.

Second, Lemma 3.2.4 indicates that for each occurring ρ-avoided word w, w_p is a path-label of an explicit node v of $\mathcal{T}(x)$. Routine OCCURRINGAVOIDEDWORDS considers every child of each such node of word-depth k, and verifies if its path-label is a ρ-avoided word. □

Lemma 3.3.2. Given a word x of length n over a fixed-sized alphabet, an integer $k > 2$, and a real number $\rho < 0$, Algorithm AVOIDEDWORDS requires time and space $\mathcal{O}(n)$; for integer alphabets, it requires time $\mathcal{O}(\sigma n)$.

Proof. Constructing the suffix tree $\mathcal{T}(x)$ of the input word x takes time and space $\mathcal{O}(n)$ for a word over a fixed-sized alphabet [22]. Once the suffix tree is constructed, computing arrays $\mathcal{D}$ and $\mathcal{C}$ by traversing $\mathcal{T}(x)$ requires time and space $\mathcal{O}(n)$. Note that the path-labels of the nodes of $\mathcal{T}(x)$ can by implemented in time and space $\mathcal{O}(n)$ as follows: traverse the suffix tree to compute for each node v the smallest index i of the terminal nodes of the subtree rooted at v. Then $\mathcal{L}(v) = x[i \mathinner{.\,.} i + \mathcal{D}(v) - 1]$.

Next, Routine ABSENTAVOIDEDWORDS requires time $\mathcal{O}(n)$. It starts by computing all minimal absent words of x, which can be achieved in time and space $\mathcal{O}(n)$ over a fixed-sized alphabet [**? ?**]. The rest of the procedure deals with checking each of the $\mathcal{O}(n)$ minimal absent words of length k. Checking each minimal absent word w to determine whether it is a ρ-avoided word or not requires time $\mathcal{O}(1)$. In particular, an $\mathcal{O}(n)$-time pre-processing of $\mathcal{T}(x)$ allows the retrieval of the (implicit or explicit) node in $\mathcal{T}(x)$ corresponding to the longest proper prefix of w in time $\mathcal{O}(1)$ [30]. Finally, Routine OCCURRINGAVOIDEDWORDS requires time $\mathcal{O}(n)$. It traverses the suffix tree $\mathcal{T}(x)$ to consider all explicit nodes of word-depth $k-1$. Then for each such node, the procedure checks every (explicit or implicit) child of word-depth k. The total number of these children is at most $n-k+1$. For every child node, the procedure checks whether its path-label is a ρ-avoided word in time $\mathcal{O}(1)$ via the use of suffix-links.

For integer alphabets, the suffix tree can be constructed in time $\mathcal{O}(n)$ [26] and all minimal absent words can be computed in time $\mathcal{O}(\sigma n)$ [**? ?**]. The efficiency of Algorithm AVOIDEDWORDS is then limited by the total number of words to be considered, which, by Lemma 3.2.5, is $\mathcal{O}(\sigma n)$. We denote q as implicit node or explicit node on suffix tree. And $q = \text{CHILD}(v, \alpha)$ means, between node v and node $Child(v, \alpha)$, every implicit node which along this edge that correspond to words (potential w_p's) whose proper longest suffix (the respective w_i) is represented by an explicit node in $\mathcal{T}(x)$, and also $q = \text{CHILD}(v, \alpha)$ considers node $Child(v, \alpha)$. Note that for integers alphabets, a batch of q $\text{CHILD}(v, \alpha)$ queries can be answered off-line in time $\mathcal{O}(n+q)$ with the aid of radix sort (in Routine ABSENTAVOIDEDWORDS) or on-line in time $\mathcal{O}(q \log \sigma)$ (in Routine OCCURRINGAVOIDEDWORDS). □

Theorem 3.3.3. Algorithm AVOIDEDWORDS solves Problem AVOIDEDWORDSCOMPUTATION in time and space $\mathcal{O}(n)$. For integer alphabets, the algorithm solves the problem in time $\mathcal{O}(\sigma n)$; this is asymptotically time-optimal if $\sqrt{n}+1 \leq \sigma \leq n$.

3.3.2 Computation of All ρ-Avoided Words

Although the biological motivation is yet to be shown for this, we present here how we can modify Algorithm AVOIDEDWORDS so that it computes *all* ρ-avoided words (of all lengths) in a given word x of length n over an integer alphabet of size σ in time $\mathcal{O}(\sigma n)$. We further show that this algorithm (ALLAVOIDEDWORDS) is in fact time-optimal.

Based on Lemma 3.2.1 and similarly to the proof of Lemma 3.2.5 we obtain the following result.

The implementation of this procedure is given in Routine ALLAVOIDEDWORDS.

Lemma 3.3.4. The number of ρ-avoided words in a word of length n over an alphabet of size $2 \leq \sigma \leq n$ is $\mathcal{O}(\sigma n)$ and this bound is tight.

Proof. By Lemma 3.2.3, every ρ-avoided word is either occurring or a minimal absent word. The set of occurring ρ-avoided words in x can be injected to the set of explicit nodes of $\mathcal{T}(x)$ by Lemma 3.2.4 and it is well known that the number of explicit nodes of $\mathcal{T}(x)$ is no more than $2n$ [22]. Moreover the number of minimal absent words of a word of length n over an alphabet of size σ is smaller than or equal to σn [47]. Hence the number of ρ-avoided words is bounded from above by $(\sigma+2)n$. This implies that asymptotically they are $\mathcal{O}(\sigma n)$.

Further, based on Lemma 3.2.1, we know that for any alphabet of size $2 \leq \sigma \leq n$ there exist words with $\Omega(\sigma n)$ minimal absent words. Consider such a word x and some $\rho \geq -\frac{1}{n}$. Then, similarly to the proof of Lemma 3.2.5, any minimal absent word w of x is ρ-avoided since $E(w) \geq \frac{1}{n}$ and $f(w) = 0$ and hence $std(w) \leq -\frac{1}{n} \leq \rho$. It follows that the bound is tight for alphabets of size $2 \leq \sigma \leq n$. □

It is clear that if we just remove the condition on the length of each minimal absent word in Line 2 of AbsentAvoidedWords we then compute all absent ρ-avoided words in time $\mathcal{O}(\sigma n)$. In order to compute all occurring ρ-avoided words in x it suffices by Lemma 3.2.4 to investigate the children of explicit nodes. We can thus traverse the suffix tree $\mathcal{T}(x)$ and for each explicit internal node, check for all of its children (explicit or implicit) whether their path-label is a ρ-avoided word. We can do this in $\mathcal{O}(1)$ time as described. The total number of these children is at most $2n-1$, as this is the bound on the number of edges of $\mathcal{T}(x)$ [22]. This modified algorithm is clearly time-optimal for fixed-sized alphabets as it then runs in time $\mathcal{O}(n)$. The time optimality for integer alphabets follows directly from Lemma 3.3.4.

Theorem 3.3.5. Algorithm AllAvoidedWords solves Problem AllAvoidedWordsComputation in time $\mathcal{O}(\sigma n)$. This is time-optimal if $2 \leq \sigma \leq n$.

Remark 3.3.1. In [29], it is shown that all $|\mathcal{A}|$ minimal absent words of a word x of length n over an integer alphabet can be computed in time $\mathcal{O}(n+|\mathcal{A}|)$ and space $\mathcal{O}(n)$.

Computing minimal absent words and checking for each of them if it is an avoided word is the bottleneck for algorithms AVOIDEDWORDS and ALLAVOIDEDWORDS. The result of [29] implies that for a word x of length n over an integer alphabet we can make both algorithms to require time $\mathcal{O}(n+|\mathcal{A}|)$ and space $\mathcal{O}(n)$. We can do that by checking for each minimal absent word output by the algorithm whether it is avoided, instead of storing a representation of them and then making the check.

Remark 3.3.2. As the complexity of algorithms AVOIDEDWORDS and ALLAVOIDEDWORDS does not depend on the value of ρ, one can use a negative ρ close to 0, sort the output ρ-avoided words with respect to $dev(w)$, and consider the extreme ones.

Lemma 3.3.6. The expected length of the longest ρ-avoided word in a word x of length n over an alphabet Σ of size $\sigma > 1$ is $\mathcal{O}(\log_\sigma n)$ when the letters are independent and identically distributed random variables uniformly distributed over Σ.

Proof. By Lemma 3.2.4 the length of the longest occurring word is bounded above by the word-depth of the deepest internal explicit node in $\mathcal{T}(x)$ incremented by 1. We note that the greatest word-depth of an internal node corresponds to the longest repeated factor in word x. Moreover, for a word w to be a minimal absent word, w_i must appear at least twice in x (in the occurrences of w_p and w_s). Hence the length of the longest ρ-avoided word is bounded by the length of the longest repeated factor in x incremented by 2. The expected length of the longest repeated factor in a word is known to be $\mathcal{O}(\log_\sigma n)$ [44] and hence the lemma follows. □

ALLAVOIDEDWORDS(x, ρ)

1 $\mathcal{T}(x) \leftarrow$ SUFFIXTREE(x)

2 **for** each node $v \in \mathcal{T}(x)$ **do**

3 $\quad\mathcal{D}(v) \leftarrow$ word-depth of v

4 $\quad\mathcal{C}(v) \leftarrow$ number of terminal nodes in the subtree rooted at v

5 ALLABSENTAVOIDEDWORDS(x, ρ)

6 ALLOCCURRINGAVOIDEDWORDS(x, ρ)

```
ALLABSENTAVOIDEDWORDS(x, ρ)

 1  A ← MINIMALABSENTWORDS(x)
 2  for each tuple <(i, j), α> ∈ A do
 3      u_p ← NODE(i, j)
 4      if ISIMPLICIT(u_p) then
 5          (u, v) ← EDGE(u_p)
 6          f_p ← C(v)
 7      else  f_p ← C(u_p)
 8      u_i ← NODE(i+1, j)
 9      if ISIMPLICIT(u_i) then
10          (u, v) ← EDGE(u_i)
11          f_i ← f_s ← C(v)
12      else  f_i ← C(u_i)
13          u_s ← CHILD(u_i, α)
14          f_s ← C(u_s)
15      E ← f_p × f_s / f_i
16      if (0 − E)/(max{1, √E}) ≤ ρ then
17          REPORT(x[i..j]α)
```

ALLOCCURRINGAVOIDEDWORDS(x, ρ)

```
1   N ← an empty stack
2   PUSH(N, root(T(x)))
3   while N is not empty do
4       u ← POP(N)
5       for each edge (u, v) of T(x) do
6           PUSH(N, v)
7       f_p ← C(v)
8       f_i ← C(suffix-link[v])
9       for each v' = CHILD(v, α), α ∈ Σ do
10          f_w ← C(v')
11          f_s ← C(CHILD(suffix-link[v], α))
12          E ← f_p × f_s / f_i
13          if (f_w − E)/(max{1, √E}) ≤ ρ then
14              REPORT(L(v')[0..n − 1])
```

3.3.3 Computation of All ρ-Overabundant words

Based on Fact 3.2.7 and Lemma 3.2.9 all ρ-overabundant words of a word x are factors of x of the form ayb, where $a,b \in \Sigma$ and y is the label of an explicit node of $\mathcal{T}(x)$. It thus suffices to consider these words and check for each of them whether it is ρ-overabundant. We can find the ones that have their longest proper prefix represented by an explicit node in $\mathcal{T}(x)$ easily, by taking the suffix-link from that node during a traversal of the tree. To find the ones that have their longest proper prefix represented by an implicit node we use the following fact, which follows directly from the definition of the suffix-links of the suffix tree.

Fact 3.3.7. Suppose aw, where $a \in \Sigma$ and $w \in \Sigma^*$, is a factor of a word x and that w is represented by an explicit node v in $\mathcal{T}(x)$, while aw by an implicit node along the edge (u_1,u_2) in $\mathcal{T}(x)$. The suffix-link from u_2 points to a node in the subtree of $\mathcal{T}(x)$ rooted at v.

The algorithm first builds the suffix tree of word x, which can be done in time and space $\mathcal{O}(n)$ for words over an integer alphabet [26]. It is also easy to compute $\mathcal{D}(v)$ and $\mathcal{C}(v)$, for each node v of $\mathcal{T}(x)$, within the same time complexity (lines $1-4$ in Algorithm OVERABUNDANTWORDS).

The algorithm then performs a traversal of $\mathcal{T}(x)$. When it first reaches a node v, it considers $\mathcal{L}(v)$ as a potential longest proper prefix of ρ-overabundant words—i.e. $\mathcal{L}(v) = w_p = aw_i$, where $a \in \Sigma$. By following the suffix-link to node u, which represents the respective w_i, and based on the first letter of the label of each outgoing

edge (v,q) from v, it computes the deviation for all possible factors of x of the form $w_p b$, where $b \in \Sigma$. (Note that we can answer all the $\textsc{Child}(u,\alpha)$ queries off-line in time $\mathcal{O}(n)$ in total for integer alphabets.) It is clear that this procedure can be implemented in time $\mathcal{O}(n)$ in total (lines $6-19$).

Then, while on node v and based on Fact 3.3.7, the algorithm considers for every outgoing edge (v,q), the implicit nodes along this edge that correspond to words (potential w_p's) whose proper longest suffix (the respective w_i) is represented by an explicit node in $\mathcal{T}(x)$.

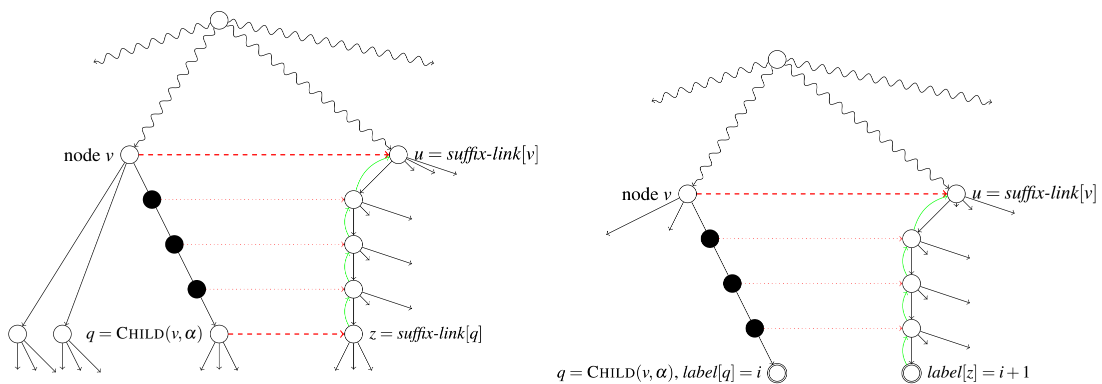


Fig. 3.1 The above figures illustrate the nodes (implicit or explicit) considered in a step (lines 6-36) of Algorithm OVERABUNDANTWORDS. *The figure on the left presents the case where* $\textsc{Child}(v,\alpha)$ *is an internal node, while the right one the case that it is a leaf. Black nodes represent implicit nodes along the edge* (v,q) *that we have to consider as potential* w_p*, and the red dotted line joins them with the respective (white) explicit node that represents the longest suffix of this* w_p*, i.e.* w_i.

Hence, when $\mathcal{D}(q) - \mathcal{D}(v) > 1$ the algorithm follows the suffix-link from node q to node z. It then checks whether $\textsc{Parent}(z) = u$. If not, then the word $\mathcal{L}(q)[0\,..\,\mathcal{D}(\textsc{Parent}(z))]$ is represented by an implicit node along the edge (v,q) and hence $\mathcal{L}(q)[0\,..\,\mathcal{D}(\textsc{Parent}(z))+1]$ has to be checked as a potential ρ-overabundant word.

After the check is completed, the algorithm sets $z = \textsc{Parent}(z)$ and iterates until $\textsc{Parent}(z) = u$. This is illustrated in Figure 3.1. By Theorem 3.2.10, the $\textsc{Parent}(z) = u$ check will fail $\mathcal{O}(n)$ times in total. All other operations take time $\mathcal{O}(1)$ and hence this procedure takes time $\mathcal{O}(n)$ in total (lines $20-36$).

We formalise this procedure in Algorithm OVERABUNDANTWORDS, where we assume that the suffix tree of $x\$$ is built, where $\$$ is a special letter, $\$ \notin \Sigma$. This forces all terminal nodes in $\mathcal{T}(x)$ to be leaf nodes. We thus obtain the following result; optimality follows directly from Lemma 3.2.12.

Theorem 3.3.8. Algorithm OVERABUNDANTWORDS solves problem ALLOVERABUNDANTWORDSCOMPUTATION in time and space $\mathcal{O}(n)$, and this is time-optimal.

```
OVERABUNDANTWORDS(x,ρ)
 1  T(x) ← BUILDSUFFIXTREE(x)
 2  for each node v ∈ T(x) do
 3      D(v) ← word-depth of v
 4      C(v) ← number of terminal nodes in the subtree rooted at v
 5  for each node v ∈ T(x)   (prefix node) do
 6      ▷ Report ρ-overabundant words w such that w_p is explicit
 7      u ← suffix-link[v]   (infix node)
 8      if D(v) > 1 and ISINTERNAL(v) then
 9          f_p ← C(v)
10          f_i ← C(u)
11          if f_i > f_p and u ≠ ROOT(T(x)) then
12              for each child y of node v do
13                  if not(ISTERMINAL(y)  and  D(y) = D(v) + 1) then
14                      f_w ← C(y)
15                      α ← L(y)[D(v) + 1]
16                      f_s ← C(CHILD(u, α))
17                      E ← f_p × f_s/f_i
18                      if (f_w − E)/(max{1, √E}) ≥ ρ then
19                          REPORT(L(y)[0..D(v)])
20      ▷ Report ρ-overabundant words w such that w_p is implicit
21      for each child y of node v do
22          if D(y) > D(v) + 1 then
23              if ISINTERNAL(y) then
24                  z ← suffix-link[y]
25              else  i ← label[y]   (y is a terminal node)
26                  z ← node[i + 1]
27                  if D(z) = D(PARENT(z)) + 1 then
28                      z ← PARENT(z)
29              f_w ← f_p ← C(y)
30              while PARENT(z) ≠ u do
31                  f_i ← C(PARENT(z))
32                  f_s ← C(z)
33                  E ← f_p × f_s/f_i
34                  if (f_w − E)/(max{1, √E}) ≥ ρ then
35                      REPORT(L(y)[0..D(PARENT(z)) + 1])
36                  z ← PARENT(z)
```

3.4 Implementation and Experiments

3.4.1 Avoided words

Algorithm AVOIDEDWORDS was implemented as a program to compute the ρ-avoided words of length k in one or more input sequences; there is an option to run Algorithm ALLAVOIDEDWORDS instead. The program was implemented in the C++ programming language and developed under GNU/Linux operating system. Our program makes use of the implementation of the compressed suffix tree available in the Succinct Data Structure Library [32]. The input parameters are a (Multi)FASTA file with the input sequence(s), an integer $k > 2$, and a real number $\rho < 0$. The output is a file with the set of ρ-avoided words of length k per input sequence. The implementation is distributed under the GNU General Public License, and it is available at http://github.com/solonas13/aw. The experiments were conducted on a Desktop PC using one core of Intel Core i5-4690 CPU at 3.50GHz under GNU/Linux. The program was compiled with g++ version 4.8.4 at optimisation level 3 (-O3). We also implemented a brute-force approach for the computation of ρ-avoided words. We mainly used it to confirm the correctness of our implementation. Here we do not plot the results of the brute-force approach as it is easily understood that it is orders of magnitude slower than our approach.

Experiment I. To evaluate the time performance of our implementation, synthetic DNA ($\sigma = 4$) and protein ($\sigma = 20$) data were used. The input sequences were generated using a randomised script. In the first experiment, our task was to establish that the performance of the program does not essentially depend on k and ρ; i.e., the elapsed time of the program remains unchanged up to some constant with increasing values of

k and decreasing values of ρ. As input datasets, for this experiment, we used a DNA and a protein sequence both of length 1M (1 Million letters). For each sequence we used different values of k and ρ. The results, for elapsed time are plotted in Fig. 3.3. It becomes evident from the results that the time performance of the program remains unchanged up to some constant. The longer time required for the protein sequences for some value of k is explained by the increased number of branching nodes in this depth in the corresponding suffix tree due to the size of the alphabet ($\sigma = 20$). To confirm this we counted the number of nodes considered by the algorithm to compute the ρ-avoided words for $k = 4$ and $\rho = -10$ for both sequences. The number of considered nodes for the DNA sequence was 260 whereas for the protein sequence it was $1,585,510$. Notice that the suffix tree of a word of length n possesses between $n+1$ and $2n$ nodes.

Experiment II. In the second experiment, our task was to establish the fact that the elapsed time and memory usage of the program grow linearly with n, the length of the input sequence. As input datasets, for this experiment, we used synthetic DNA and proteins sequences ranging from 1 to 128 M. For each sequence we used constant values for k and ρ: $k = 8$ and $\rho = -10$. The results, for elapsed time and peak memory usage, are plotted in Fig. 3.6. It becomes evident from the results that the elapsed time and memory usage of the program grow linearly with n. The longer time required for the protein sequences compared to the DNA sequences for increasing n is explained by the increased number of branching nodes in this depth ($k = 8$) in the corresponding suffix tree due to the size of the alphabet ($\sigma = 20$). To confirm this we counted the number of nodes considered by the algorithm to compute the ρ-avoided words for $n = 64$M for

both the DNA and the protein sequence. The number of nodes for the DNA sequence was $69,392$ whereas for the protein sequence it was $43,423,082$.

Experiment III. In the next experiment, our task was to evaluate the time and memory performance of our implementation with real data. As input datasets, for this experiment, we used all chromosomes of the human genome. Their lengths range from around 46M (chromosome 21) to around 249M (chromosome 1). For each sequence we used $k = 8$ and $\rho = -10$. The results, for elapsed time and peak memory usage, are plotted in Fig. 3.5. The results with real data confirm that the elapsed time and memory usage of the program grow linearly with n.

Experiment IV. In an experiment with a prokaryote, we computed the set of avoided words for $k = 6$ (hexamers) and $\rho = -10$ in the complete genome of *E. coli* and sorted the output in increasing order of their deviation. The most avoided words were extremely enriched in self-complementary (palindromic) hexamers. In particular, within the output of 28 avoided words, 23 were self-complementary; and the 17 most avoided ones were *all* self-complementary. For comparison, we computed the set of avoided words for $k = 6$ and $\rho = -10$ from an eukaryotic sequence: a segment of the human chromosome 21 (its leftmost segment devoid of `N`'s) equal to the length of the *E. coli* genome. In the output of 10 avoided words, no self-complementary hexamer was found. Our results confirm that the restriction endonucleases which target self-complementary sites are not found in eukaryotic sequences [55].

Experiment V. Then, we proceeded to the examination of several collections of CNEs obtained through multiple sequence alignment between the human and other genomes. The detailed description of how those CNEs were identified could be found

in [52]. For each CNE of these datasets, a sequence stretch (surrogate sequence) of non-coding DNA of equal length and equal GC content was taken at random from the repeat-masked human genome. The CNEs of each collection were concatenated into a single long sequence and the same procedure was followed for the corresponding surrogates. Seven CNEs concatenates and the corresponding surrogate datasets have been formed and used in this experiment. We have determined through the proposed algorithm the avoided words for $k = 10$ (decamers) and $\rho = -2$ for these fourteen datasets and the results are presented in Table 3.1. In Table 3.2, we show likewise for $k > 2$ (all avoided words) and $\rho = -2$.

The first five CNEs collections have been composed through multiple sequence alignment of the same set of genomes and they differ only in the thresholds of sequence similarity applied between the considered genomes: from 75-80 (the least conserved CNEs, which thus are expected to serve less demanding functional roles) to 95-100 which represent the extremely conserved non-coding elements (UCNEs or CNEs 95-100) [52]. The remaining two collections have been composed under different constraints and have been derived after alignment of genomes belonging to the *Mammalian* and *Amniotic* groups. In Tables 3.1 and 3.2, the last line shows the ratios formed by the numbers of avoided words of each concatenate of surrogates divided by the numbers of avoided words of the corresponding CNE dataset.

Two immediate results stem from inspection of Tables 3.1 and 3.2:

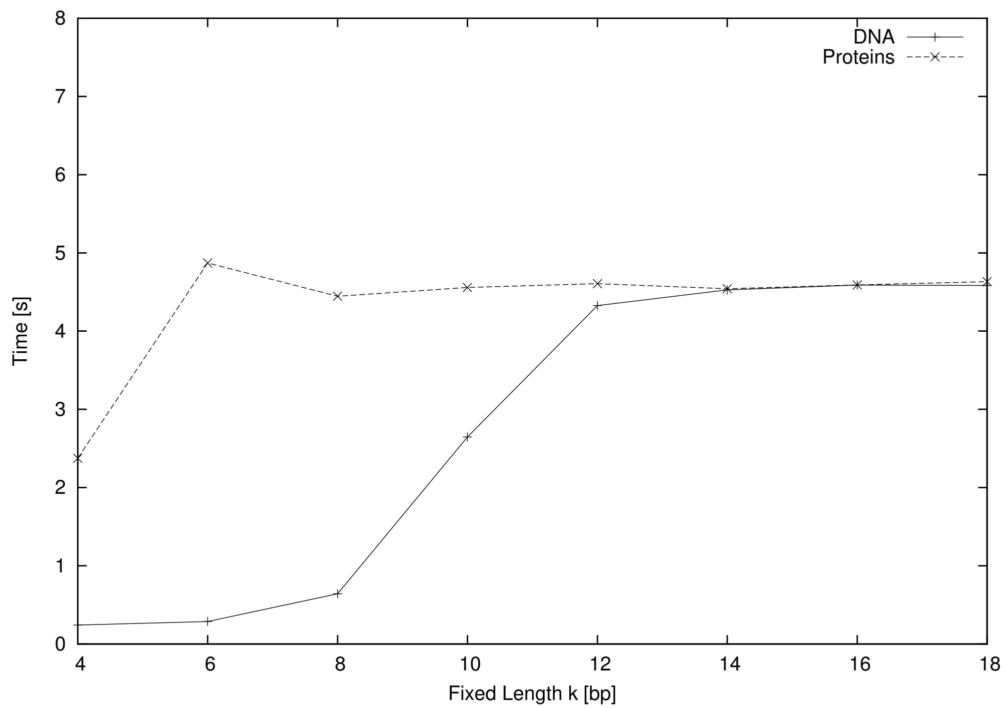
1. In all cases, the number of avoided words from the non-functional (surrogate) concatenate of sequences far exceeds the corresponding number derived from the corresponding CNE dataset.

2. In the case of datasets with increasing degree of similarity between aligned genomes (from 75-80 to 95-100) the ratios of the numbers of avoided words show a clear increasing trend.

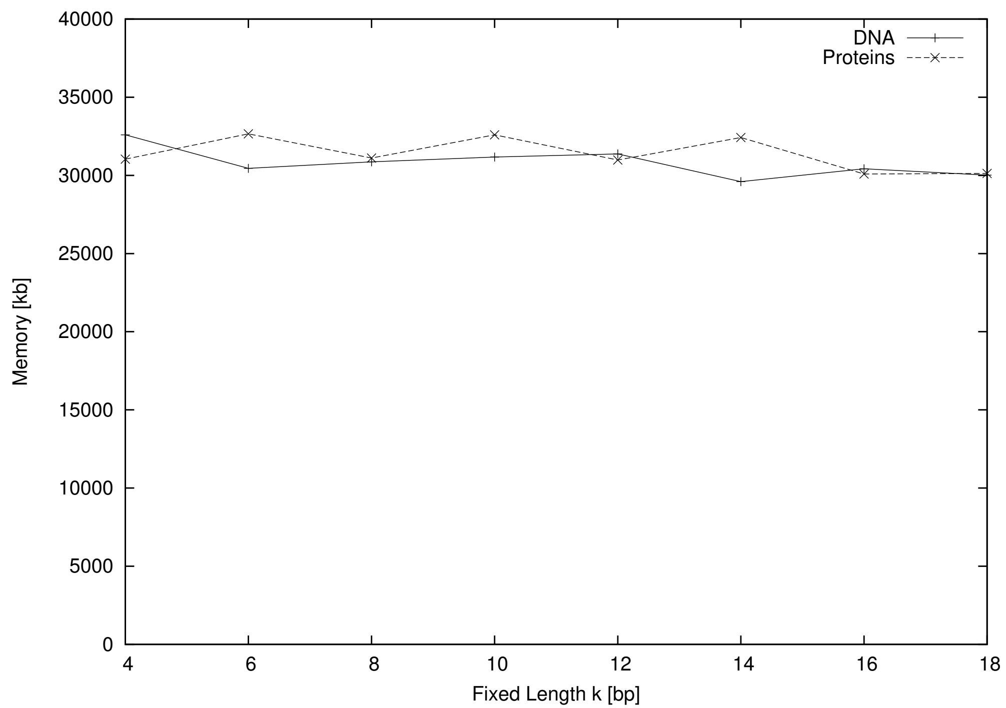
Both these findings can be understood on the basis of the difference in functionality, and thus tolerance to mutations, between CNE and surrogate datasets. One particularly frequent source of mutations is the slippage error during DNA replication; see e.g. reference [35]. Within a genomic sequence, this phenomenon causes the generation and increase in length, during evolutionary time, of polypyrimidine and polypurine nucleotide tracts. The expansion of those tracts is impeded at a considerable degree in the case of sequences which serve a functional role (as CNEs do) due to several constraints. On the other hand, in non-functional regions (as our surrogates mostly are) this procedure ceases to be tolerated only when it reaches to the formation of a polypyrimidine/polypurine tract with length affecting the proper folding or other structural features of the chromatin. Then, selection eliminates it, while its longer proper factors are tolerated in sufficient numbers within the sequence, thus resulting to an avoided word. In support of this explanation is the observation that all lists of avoided words found by our algorithm in concatenates of surrogates exhibit a considerable enrichment in oligopurines and oligopyrimidines. Taking at random some examples, for $k = 10$, we notice: `AAAAAAAAAT`, `AAAAAACCAC`, `ACAAAAAAAA`, `CTCCTCTTTT`, etc.

Our second observation, i.e. the positive correlation between (i) the paucity of avoided decamers in CNEs collections and (ii) the similarity thresholds used for their identification comes in accordance with the above argument. CNEs extracted under a stricter requirement of sequence similarity between evolutionary distant species are

CNEs whose functionality is less tolerant to alterations due to random mutations in general. Hence, they also tolerate less the propagation within their sequence of parasite polypyrimidine/polypurine tracts too.

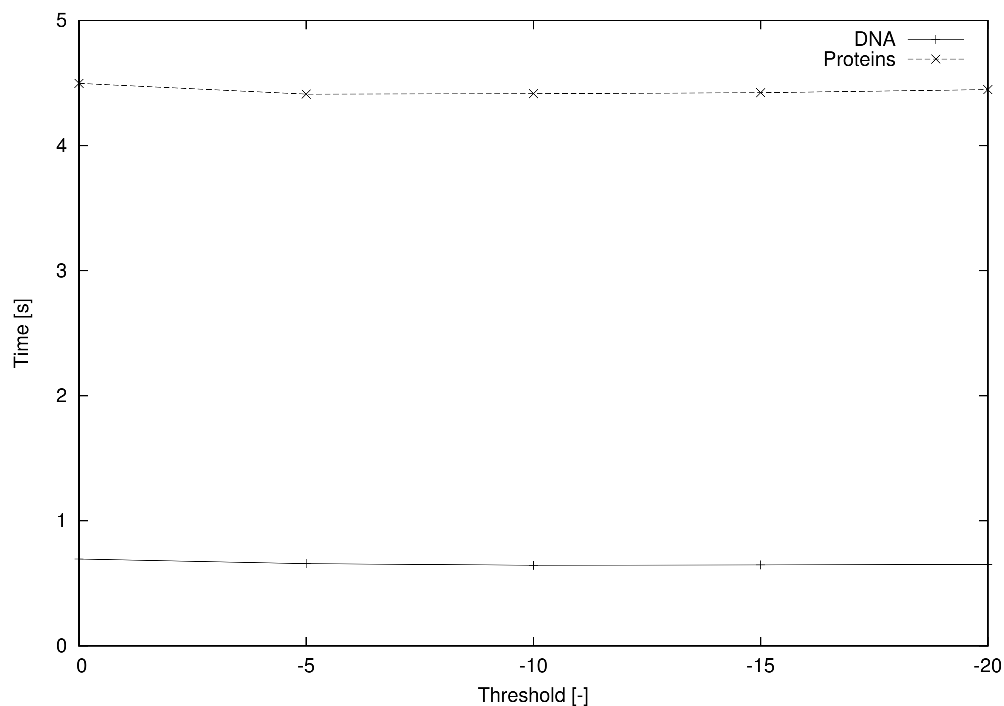


(a) Time for $n = 1$Mbp and $\rho = -10$

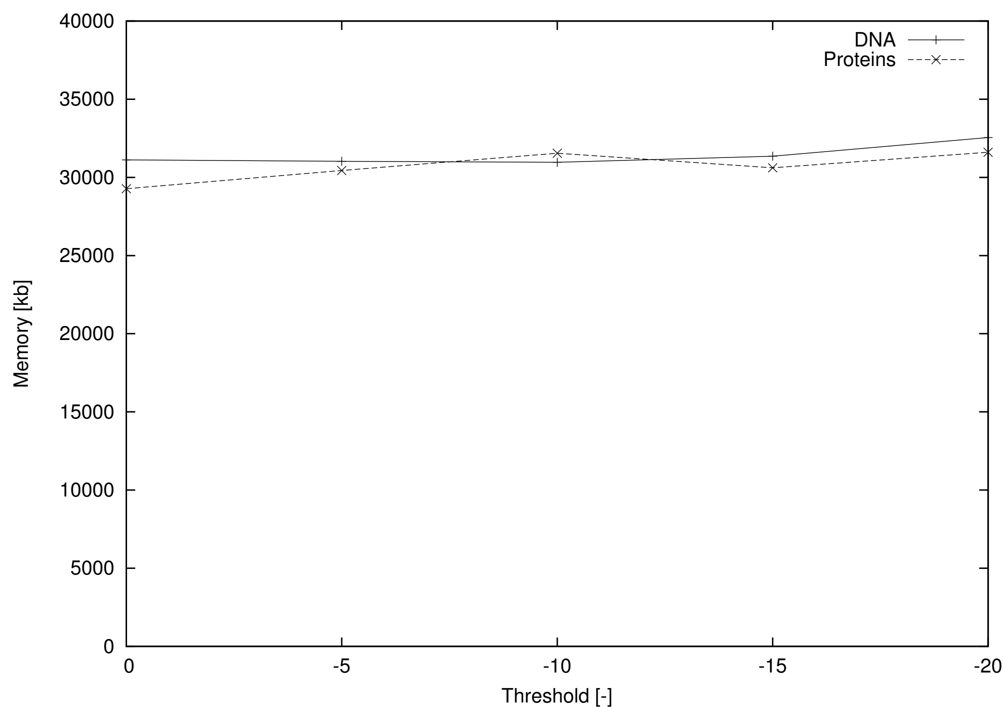


(b) Memory for $n = 1$Mbp and $\rho = -10$

Fig. 3.2 Experiment I. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS *using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) data of length* 1*MB for variable k.*

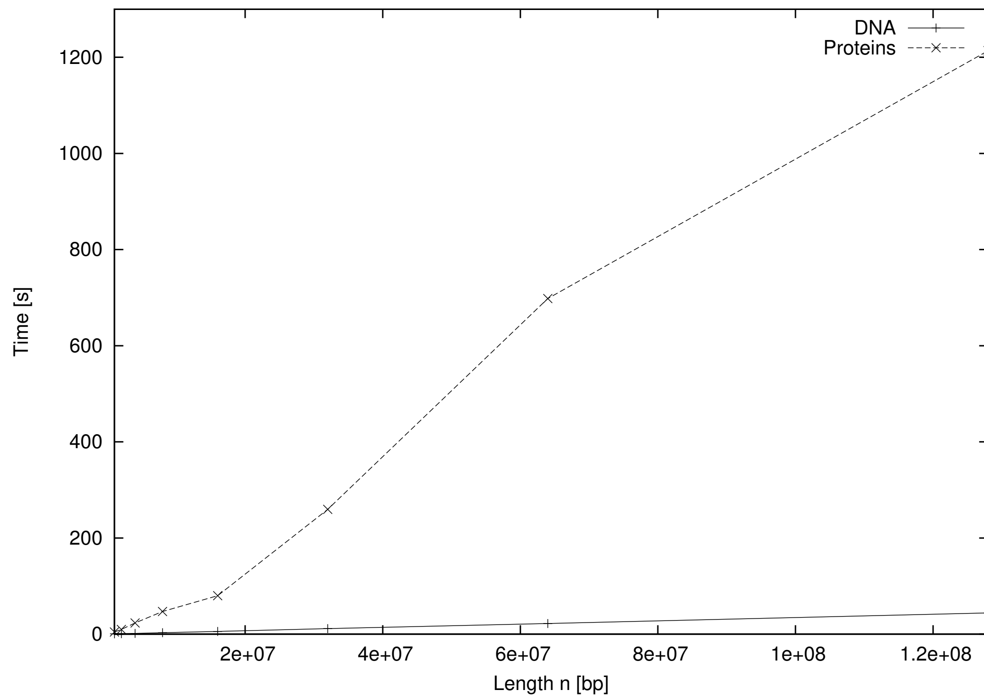


(a) Time for $n = 1$Mbp and $k = 8$

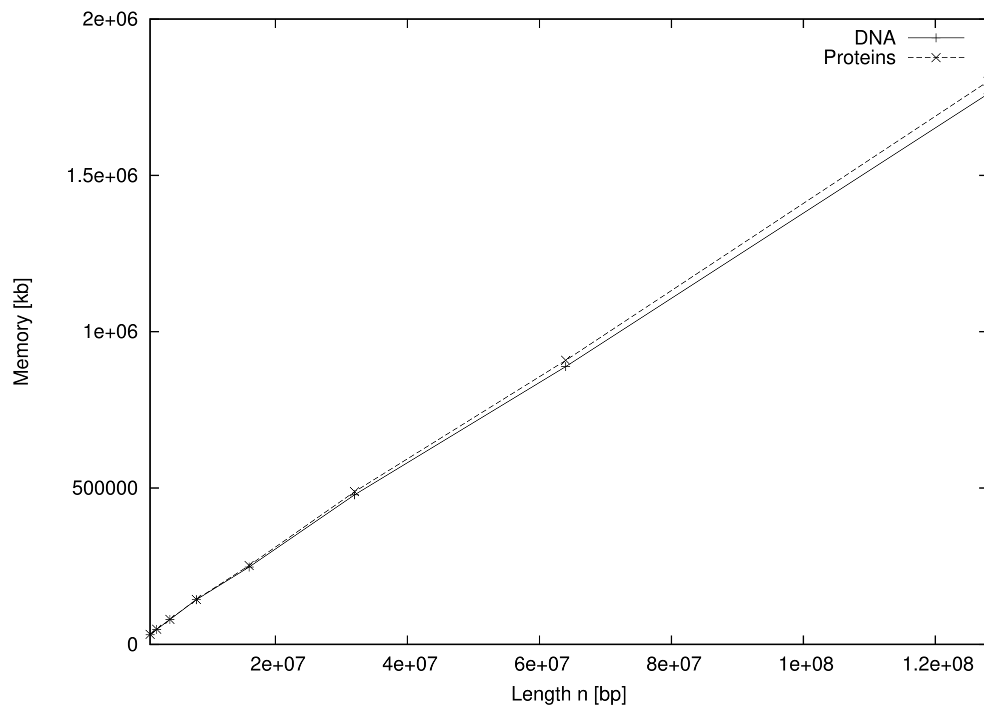


(b) Memory for $n = 1$Mbp and $k = 8$

Fig. 3.3 Experiment I. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS *using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) data of length* 1*MB for variable ρ.*

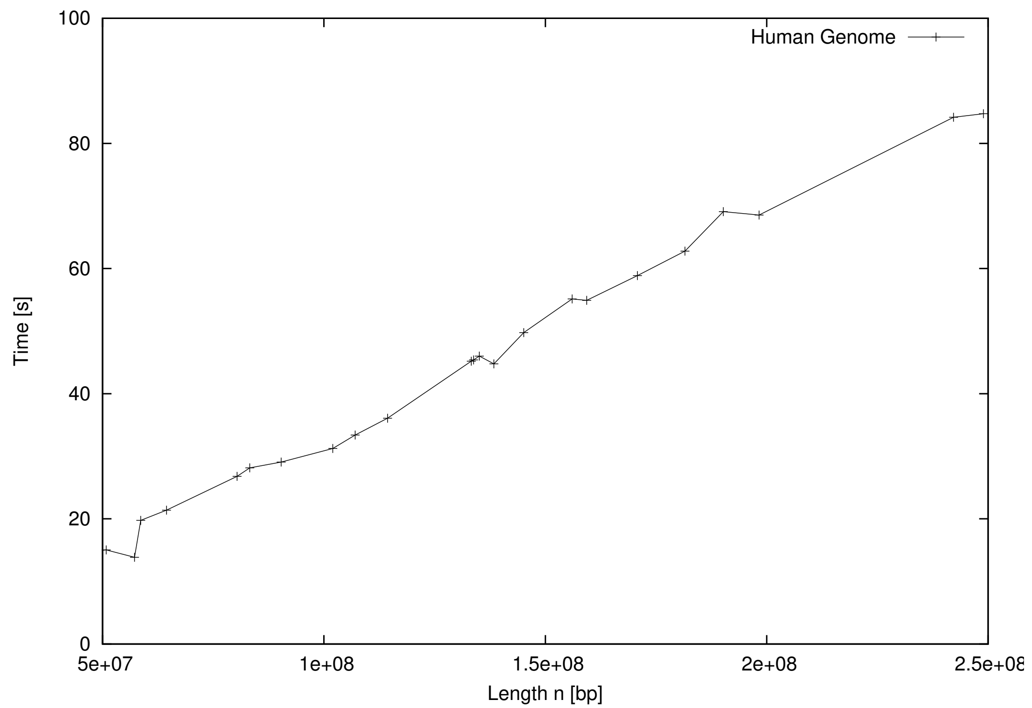


(a) Time for $k = 8$ and $\rho = -10$

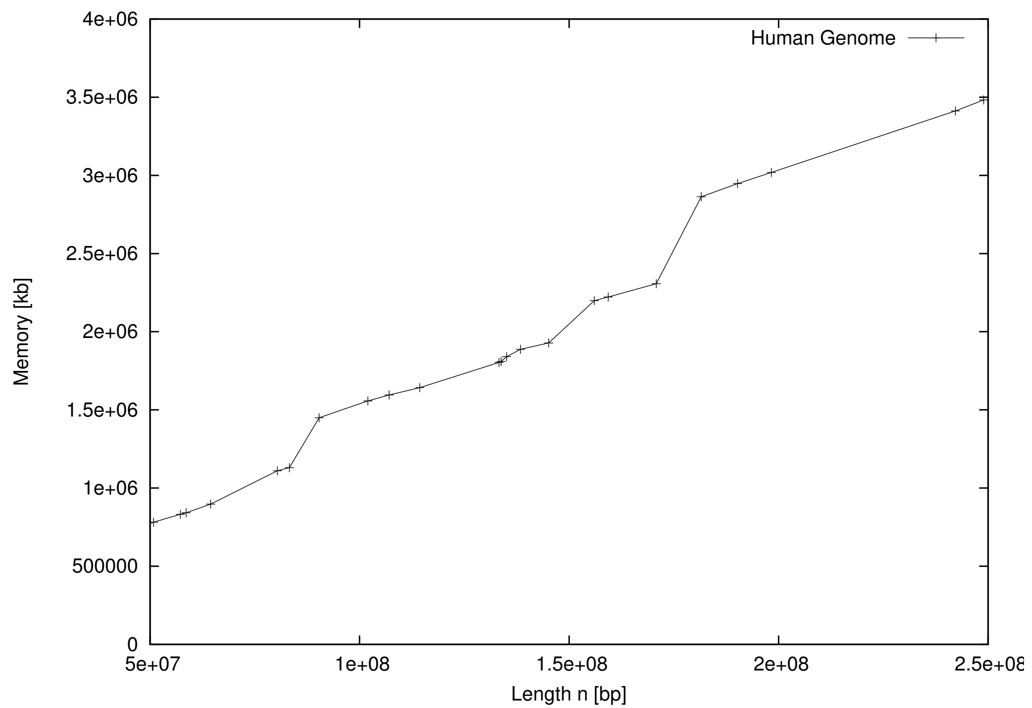


(b) Memory for $k = 8$ and $\rho = -10$

Fig. 3.4 Experiment II. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS *using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) data of length* 1*Mbp to* 128*Mbp.*



(a) Time for $k = 8$ and $\rho = -10$



(b) Memory for $k = 8$ and $\rho = -10$

Fig. 3.5 Experiment III. Elapsed time and peak memory usage of Algorithm AVOIDEDWORDS *using all chromosomes of the human genome.*

	CNEs 75-80	CNEs 80-85	CNEs 85-90	CNEs 90-95	CNEs 95-100	Mammalian	Amniotic
Surr.	1,658	810	445	256	429	29,677	6,043
CNE	514	153	51	40	45	2,821	623
Ratio	3.23	5.29	8.73	6.40	9.53	10.52	9.70

Table 3.1 The number of avoided words, for $k = 10$ and $\rho = -2$, for each concatenate of surrogates (Row 1); the number of avoided words of the corresponding CNE dataset (Row 2); and their ratio (Row 3).

	CNEs 75-80	CNEs 80-85	CNEs 85-90	CNEs 90-95	CNEs 95-100	Mammalian	Amniotic
Surr.	10,734	7,202	5,351	3,849	4,540	112,181	22,595
CNE	3,207	1,847	1,296	1,043	1,030	17,685	3,635
Ratio	3.35	3.90	4.13	3.69	4.41	6.34	6.22

Table 3.2 The number of avoided words, for $k > 2$ and $\rho = -2$, for each concatenate of surrogates (Row 1); the number of avoided words of the corresponding CNE dataset (Row 2); and their ratio (Row 3).

3.4.2 Overabundant words

Algorithm OVERABUNDANTWORDS was implemented as a program to compute the ρ-overabundant words in one or more input sequences. The program was implemented in the C++ programming language and developed under GNU/Linux operating system. Our program makes use of the implementation of the *compressed suffix tree* available in the Succinct Data Structure Library [32]. The input parameters are a (Multi)FASTA file with the input sequence(s) and a real number $\rho > 0$. The output is a file with the set of ρ-overabundant words per input sequence. The implementation is distributed under the GNU General Public License, and it is available at http://github.com/solonas13/aw. The experiments were conducted on a Desktop PC using one core of Intel Core i5-4690 CPU at 3.50GHz under GNU/Linux. The program was compiled with g++ version 4.8.4 at optimisation level 3 (-O3). We also implemented a brute-force approach to confirm the correctness of our implementation. Here we do not plot the results of the brute-force approach as it is easily understood that it is orders of magnitude slower than our linear-time approach.

Experiment I. (Effectiveness) In the first experiment, our task was to establish the effectiveness of the statistical model in identifying overabundant words. To this end, we generated 25 random sequences of length $n = 80,000$ over the DNA alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ (uniform distribution). Then for each of these sequences, we inserted a random word w of length $m = 6$ in t random positions. We varied the value of t based on the fact that in a random sequence of length n over an alphabet of size $\sigma = |\Sigma|$, where letters are independent, identically uniformly distributed random variables, a specific word of length m is expected to occur roughly $r = n/\sigma^m$ times. We hence considered t

equal to r, $2r$, $4r$, $8r$, and $16r$. We then ran our program for each resulting sequence to identify the ρ-overabundant words with $\rho = 0.000001$, and output the deviation of the inserted word w, as well as the word $w_{\max}$ with the maximum deviation. The inserted word w was reported as a ρ-overabundant word in *all* cases. Furthermore, in many cases the word with the maximum deviation was w itself and in many other cases one of its factors; this was true in *all* cases for $t \geq 80 \approx 4r$. Hence, the model is effective in identifying words that are overabundant. The full results of this experiment are presented in Table 3.3.

Experiment II. (Efficiency) Our task here was to establish the fact that the elapsed time of the implementation grows linearly with n, the length of the input sequence. As input datasets, for this experiment, we used synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) sequences ranging from 1 to 128 M (Million letters). For each sequence we used a constant value of $\rho = 10$. The results are plotted in Fig. 3.6. It becomes evident from the results that the elapsed time of the program grows linearly with n. The longer time required for the proteins sequences compared to the DNA sequences for increasing n is explained by the dependence of the time required to answer queries of the form $\textsc{Child}(v, \alpha)$ on the size of the alphabet ($\sigma = 20$ *vs.* $\sigma = 4$) in the implementation of the compressed suffix tree we used.

Experiment III. (Real Application) Here we proceed to the examination of seven collections of Conserved Non-coding Elements (CNEs) obtained through multiple sequence alignment between the human and other genomes. Despite being located at the non-coding part of genomes, CNEs can be extremely conserved on the sequence level across organisms. Their genesis, functions and evolutionary dynamics still remain

enigmatic [34, 51]. The detailed description of how those CNEs were identified can be found in [**?**]. For each CNE of these datasets, a sequence stretch (surrogate sequence) of non-coding DNA of equal length and equal GC content was taken at random from the repeat-masked human genome. The CNEs of each collection were concatenated into a single long sequence and the same procedure was followed for the corresponding surrogates. We have determined through the proposed algorithm the overabundant words for $k = 10$ (decamers) and $\rho = 3$ for these fourteen datasets and the results are presented in Table 3.4. Likewise, in Table 3.5, we show all overabundant words (i.e. $k > 2$) for $\rho = 3$.

The first five CNE collections have been composed through multiple sequence alignment of the same set of genomes (human vs. chicken; mapped on the human genome) and they differ only in the thresholds of sequence similarity applied between the considered genomes: from 75% to 80% (the least conserved CNEs, which thus are expected to serve less demanding functional roles) to 95–100% which represent the extremely conserved non-coding elements (UCNEs or CNEs 95–100) [**?**]. The remaining two collections have been composed under different constraints and have been derived after alignment of Mammalian and Amniotic genomes. In Tables 3.4 and 3.5, the last line shows the ratios formed by the numbers of overabundant words of each concatenate of surrogates divided by the numbers of overabundant words of the corresponding CNE dataset.

Inspecting data contained in Tables 3.4 and 3.5, first we observe in all cases that absolute numbers of overabundant words drop from low- to high-conserved CNE concatenates. This feature is shared by the corresponding concatenates of surrogate

sequences as evidenced along table rows from CNEs 75-80 to CNEs 95-100. This is due to the considerable decrease in absolute numbers of the corresponding elements in the human genome, which is reflected to the length of their concatenates. Note that in genomic sequences, extreme conservation is always clearly less frequent than medium conservation. As the studied sequences decrease in length, the numbers of overabundant words also drop in each category (CNEs or surrogates). Consequently, the important quantity is the ratio of these numbers between CNE and surrogate dataset. As amniotic and mammalian CNEs are classes characterized by different conservation thresholds (the former being much more conserved), they also present disparate overabundant word numbers, again the corresponding ratios being the relevant quantities.

Two results directly related to our analysis stem from inspection of Tables 3.4 and 3.5:

1. In all cases, the number of overabundant words from the surrogate concatenate of sequences *far exceeds* the corresponding number derived from the CNE dataset.

2. In the case of datasets with increasing degree of similarity between aligned genomes (from 75-80 to 95-100), the ratios of the numbers of overabundant words show a clear, *increasing trend*.

Both these findings can be understood on the basis of the difference in functionality between CNE and surrogate datasets. As we briefly describe in the introduction, this systematic difference (finding 1 above) is expected on the basis of the self-enhancing elongation of relatively long homonucleotide tracts [41**?**], which occurs mainly in the non-constrained parts of the genome, here the surrogate datasets. Therefore, we expect and we do find that CNE datasets always have less overabundant words than their

corresponding surrogate. Moreover, finding 2 corroborates the proposed mechanism of overabundance, as in CNE datasets 1-5 depletion in overabundant words quantitatively follows the degree of sequence conservation. Inspection of the individual overabundant words found in the surrogate datasets verifies that they largely consist of short repeats of the types described in [?] and in [41]. There is an analogy of this finding with a corresponding one, concerning the occurrence of avoided words in the same sequence sets, which is described in [3].

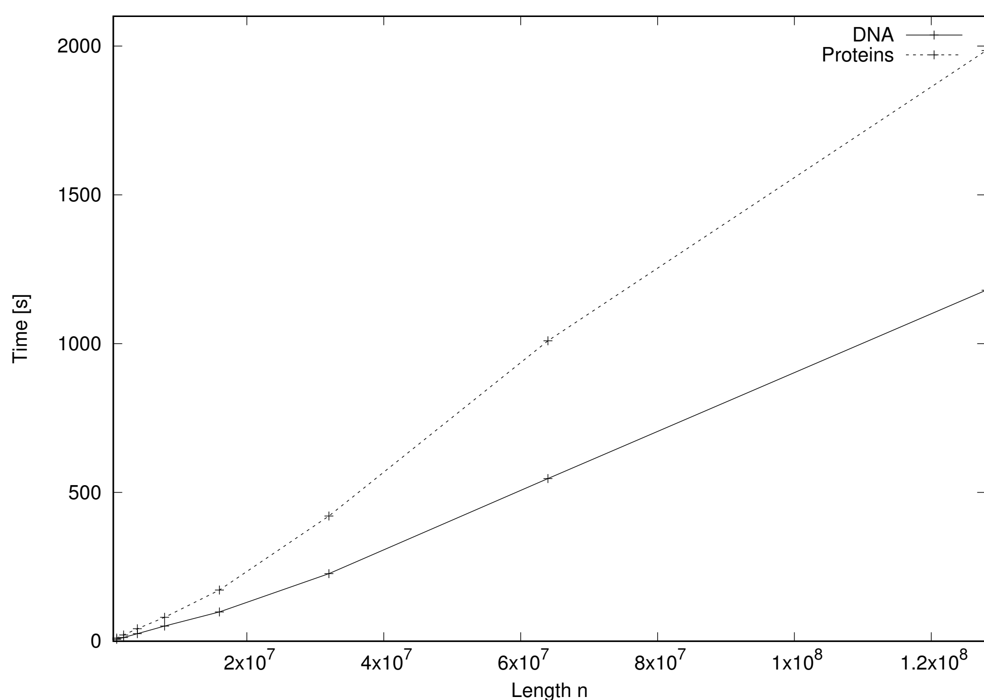


Fig. 3.6 Elapsed time of Algorithm OVERABUNDANTWORDS *using synthetic DNA ($\sigma = 4$) and proteins ($\sigma = 20$) sequences of length* $1M$ *to* $128M$.

Times t of inserting w	20	40	80	160	320
w	TTACAA	GTGCCC	CACTTT	AGTTAC	AAACAG
$std(w)$	2.233313	4.143015	5.623615	6.010327	5.674220
$w_{\max}$	CTCCTATG	GTGCCC	CACTTT	AGTTA	ACAG
$std(w_{\max})$	3.354102	4.143015	5.623615	6.900740	9.617803
w	AATCTG	AGTCGA	GAAGTC	TATCTT	CAAAAA
$std(w)$	2.034233	2.888529	4.456468	5.073860	11.071170
$w_{\max}$	ATTGGGG	TCTGTATG	GAAGTC	ATCTT	CAAAAA
$std(w_{\max})$	3.265609	3.272727	4.456468	6.115612	11.071170
w	GTACCA	GGCGTG	AAGGAT	GGGTCC	TTCCGG
$std(w)$	2.187170	3.658060	4.428189	5.467296	5.256409
$w_{\max}$	TCTGTGCG	ACGATACC	AAGGAT	GGTCC	TTCCG
$std(w_{\max})$	3.548977	4.000000	4.428189	6.787771	9.105009
w	CCATAG	GTTGAT	TGAGCG	ACATTT	CTTGTA
$std(w)$	2.470681	2.467858	4.214544	5.755475	5.362435
$w_{\max}$	CAGTGGTC	TTTTCCT	TGAGC	ACATT	TTGTA
$std(w_{\max})$	3.333333	3.368226	5.072968	6.376277	9.467110
w	TCGACA	CGCTTT	TACAAC	TATTAG	TGAGAT
$std(w)$	1.531083	2.789220	3.552902	4.959926	5.124976
$w_{\max}$	CTTTGCT	ATTACC	ACAAC	ATTAG	GACAT
$std(w_{\max})$	3.308195	3.322163	5.653479	6.837628	10.012316

Table 3.3 The deviation of the randomly generated inserted word w, as well as the word $w_{\max}$ with the maximum deviation. The length of each of the 25 randomly generated sequences over $\Sigma = \{A, C, G, T\}$ was $n = 80{,}000$, the length of w was $m = 6$, and $\rho = 0.000001$. In green are the cases when the word with the maximum deviation was w itself or one of its factors.

$k = 10$, $\rho = 3$	CNEs 75-80	CNEs 80-85	CNEs 85-90	CNEs 90-95	CNEs 95-100	Mammalian	Amniotic
Surr	1,144	718	473	297	469	15,470	2,874
CNEs	331	181	100	59	71	491	149
Ratio	**3.46**	**3.97**	**4.73**	**5.03**	**6.61**	**31.51**	**19.29**

Table 3.4 Number of overabundant words for $k = 10$ and $\rho = 3$.

$k > 2$, $\rho = 3$	CNEs 75-80	CNEs 80-85	CNEs 85-90	CNEs 90-95	CNEs 95-100	Mammalian	Amniotic
Surr	5,925	3,798	2,770	1,948	2,405	69,022	12,913
CNEs	1,373	778	512	390	403	7,549	1,401
Ratio	**4.32**	**4.88**	**5.41**	**4.99**	**5.97**	**9.14**	**9.22**

Table 3.5 Number of overabundant words for $k > 2$ and $\rho = 3$.

3.5 Conclusion

In this section we present a summary of what has been achieved. The systematic search for avoided words is particularly useful for biological sequence analysis. We presented a linear-time and linear-space algorithm for the computation of avoided words of length k in a given sequence x. We suggested a modification to this algorithm so that it computes all avoided words of x, irrespective of their length, within the same time complexity. We also presented combinatorial results with regards to avoided words and absent words. Moreover, we presented an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm for computing all overabundant words in a sequence x of length n over an integer alphabet. Our main result is based on a new *non-trivial* combinatorial property of the suffix tree $\mathcal{T}$ of x: the number of distinct factors of x whose longest infix is the label of an explicit node of $\mathcal{T}$ is no more than $3n-4$. We further show that the presented algorithm is time-optimal by proving that $\mathcal{O}(n)$ is a tight upper bound for the number of overabundant words.

At last, we made available an implementation of our algorithm. Experimental results, using both real and synthetic data, show its effectiveness and efficiency in biological sequence analysis.

Chapter 4

Maximal Palindromes

In this chapter, we consider a special type of uncertain sequence called weighted string. In a weighted string every position contains a subset of the alphabet and every letter of the alphabet is associated with a probability of occurrence such that the sum of probabilities at each position equals 1. Usually a cumulative weight threshold $1/z$ is specified, and one considers only strings that match the weighted string with probability at least $1/z$. We generalize Alatabbi *et al.*'s solution for standard strings [2] to compute maximal palindromes of a weighted string.

This chapter is organised as follows.

In Section 4.1 we introduce the background and contributions of maximal palindromes, that show the motivation and most recent work on maximal palindromes.

In Section 4.2 we present the preliminaries, and give the definition and useful properties of maximal palindromes.

In Section 4.3.1 we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm, where n is the length of the weighted string and $1/z$ is the given threshold, to compute a

smallest maximal z-palindromic factorization of a weighted string. This factorization has applications in hairpin structure prediction in a set of closely-related DNA or RNA sequences. Along the way, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute all maximal z-palindromes in weighted strings.

In Section 4.3.2 we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute a longest z-palindromes array in weighted strings.

In Section 4.4 we make available an implementation of our algorithm, using synthetic data, show the efficiency of our implementation.

Finally, in Section 4.5, we give the conclusion of maximal palindromes.

4.1 Background and Contributions

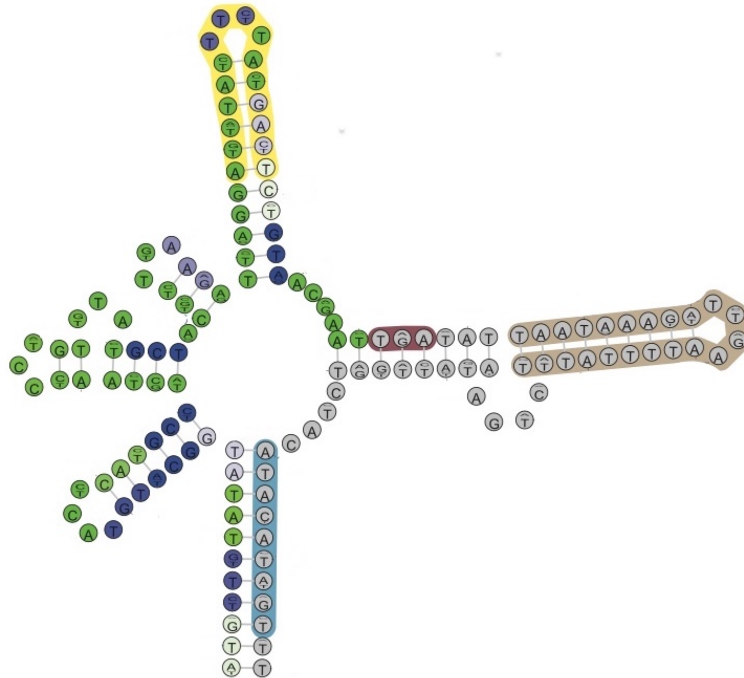
4.1.1 Background

A palindrome is a sequence that reads the same from left to right and from right to left. Detection of palindromic factors in texts is a classical and well-studied problem in algorithms on strings and combinatorics on words with a lot of variants arising out of different practical scenarios [8] [46] [53] [43] [25]. In molecular biology, for instance, palindromic sequences are extensively studied: they are often distributed around promoters, introns, and untranslated regions, playing important roles in gene regulation and other cell processes (see e.g. [3]). In particular these are strings of the form $s\bar{s}^R$, also known as complemented palindromes, occurring in single-stranded DNA or, more commonly, in RNA, where s is a string and $\bar{s}^R$ is the reverse complement of s. In DNA, `C`-`G` are complements and `A`-`T` are complements; in RNA, `C`-`G` are complements

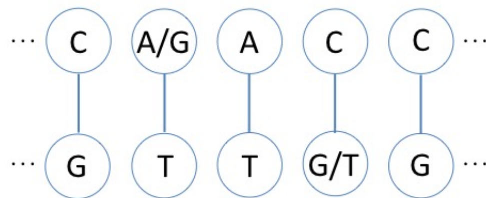
and A-U are complements. For example, AGTACTTCATGA is a standard palindrome and TAGTCGACTA is a complemented palindrome.

A string $x = x[0]x[1]\dots x[n-1]$ is said to have an initial palindrome of length k if its prefix of length k is a palindrome. Manacher first discovered an on-line algorithm that finds all initial palindromes in a string [43]. Later Apostolico *et al* observed that the algorithm given by Manacher is able to find all maximal palindromic factors in the string in $\mathcal{O}(n)$ time [8]. Gusfield gave an off-line linear-time algorithm to find all maximal palindromes in a string and also discussed the relation between biological sequences and gapped palindromes (i.e. strings of the form $sv\bar{s}^R$ where the complemented palindromes are separated by v) [33]. Searching for gapped palindromes has also been considered and efficient algorithms for this computation are known [40].

The problem that gained significant attention recently is the factorization of a string x of length n into a sequence of palindromes. We say that $x_1, x_2, \dots, x_\ell$ is a (maximal) palindromic factorization of string x, if every x_i is a (maximal) palindrome, $x = x_1 x_2 \dots x_\ell$, and ℓ is minimal, which means the number of x_i is minimal. In biological applications we need to factorise a sequence into palindromes in order to identify hairpins, patterns that occur in single-stranded DNA or, more commonly, in RNA. Alatabbi *et al.* gave an off-line $\mathcal{O}(n)$-time algorithm for finding a maximal palindromic factorization of x [2]. Fici *et al.* presented an on-line $\mathcal{O}(n \log n)$-time algorithm for computing a palindromic factorization of x [27]; a similar algorithm was presented by I *et al.* [37]. In addition, Rubinchik and Shur [54] devised an $\mathcal{O}(n)$-sized data structure that helps locating palindromes in x; they also showed how it can be used to compute a palindromic factorization of x in $\mathcal{O}(n \log n)$ time.



(a) Hairpins common to *Malvastrum yellow vein* virus, *Cotton leaf curl Multan* virus isolate, and *Bhendi yellow vein India* virus; figure taken from [48].



(b) Hairpin represented as a weighted string: `C[(A,0.5),(G,0.5)]ACC` (top) and `GTT[(G,0.5),(T,0.5)] G` (bottom).

Fig. 4.1 Hairpins that are common to a set of closely-related sequences can be represented compactly as weighted strings.

In this work, we consider a special type of uncertain sequence called weighted string (also known as position weight matrix or PWM). In a *weighted string* X every position contains a subset of the alphabet and every letter of the alphabet is associated with a probability of occurrence such that the sum of probabilities at each position equals 1. For example, we write $X = \mathtt{a[(a,0.5),(b,0.5)]}\ldots$ to denote that the probability of occurrence of a at the first position is 1 while at the second one is $1/2$, and so on. X thus represents many different strings, each with probability of occurrence equal to the *product* of probabilities of its letters at subsequent positions of X. A great deal of research has been conducted on weighted strings for indexing [13, 38], for alignments [5, 23], for pattern matching [14, 15, 39], and for finding regularities [11, 16].

4.1.2 Contributions

Muhire *et al.* [48] showed how a set of virus species can be clustered using multiple sequence alignment (MSA) to obtain subsets of viruses that have common hairpin structure (see Fig. 4.1(a)). A more compact representation of an MSA can be trivially obtained using weighted strings (see Fig. 4.1(b)). The non-trivial computational problem thus arising is how to factorize a weighted string in a sequence of palindromes.

Usually a *cumulative weight threshold* $1/z$ is specified, and one considers only strings that match the weighted string with probability at least $1/z$. We generalize Alatabbi *et al.*'s solution for standard strings [2] to compute maximal palindromes of a weighted string. In particular, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm, where n is the length of the weighted string and $1/z$ is the given threshold. Along the way, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm for computing all maximal

palindromes in weighted strings. Moreover, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute a longest z-palindromes array in weighted strings.

4.2 Preliminaries

4.2.1 Definitions and Notations

We denote the *reversal* of x by string x^R, i.e. $x^R = x[n-1]x[n-2]\ldots x[0]$. The *concatenation* of two strings x and y is the string of the letters of x followed by the letters of y. It is denoted by $x.y$ or, more simply, by xy.

A string w is said to be a *palindrome* if and only if $w = w^R$. If factor $x[i\,..\,j]$, $0 \leq i \leq j \leq n-1$, of string $x[0\,..\,n-1]$ is a palindrome, then $\frac{i+j}{2}$ is the *center* of $x[i\,..\,j]$ in x and $\frac{j-i+1}{2}$ is the *radius* of $x[i\,..\,j]$. Moreover, $x[i\,..\,j]$ is called a *palindromic factor*. It is said to be a *maximal palindrome* if there is no other palindrome in w with center $\frac{i+j}{2}$ and larger radius. A maximal palindrome w can be encoded as a pair (c,r), where c is the center of w and r is the radius of w. By $\mathcal{MP}(x)$, we denote the set of center-distinct maximal palindromes of string x. The sequence $x_1, x_2, \ldots, x_\ell$ of ℓ non-empty strings is a *(maximal) palindromic factorization* of a string x if all strings x_i are (maximal) palindromes, $x = x_1 x_2 \ldots x_\ell$, and ℓ is minimal. Note that any single letter is a palindrome and, hence, every string can always be factorized into palindromes. However, not every string can be factorized into maximal palindromes; e.g. consider $x = \texttt{abaca}$, we could not find any maximal palindrome factorization on abaca.

Example 4.2.1. Given a string $x = \texttt{abacbabcbb}$ of length 10, a maximal palindromic factorization of x is $x[0\,..\,2], x[3\,..\,7], x[8\,..\,9] = \texttt{aba,cbabc,bb}$.

Definition 4.2.1. A *weighted string* X on an alphabet Σ is a finite sequence of n sets. Every $X[i]$, for all $0 \le i < n$, is a set of ordered pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having letter s_j at position i. Formally, $X[i]$=$\{(s_j, \pi_i(s_j)) \mid s_j \neq s_l \text{ for } j \neq l, \text{ and } \Sigma\pi_i(s_j) = 1\}$. A letter s_j *occurs* at position i of X if and only if the occurrence probability of letter s_j at position $i, \pi_i(s_j)$, is greater than 0.

Note that for clarity we use upper case letters for weighted strings, e.g. X, and lower case letters, e.g. x, for standard strings.

Definition 4.2.2. A string u of length m is a *factor* of a weighted string X if and only if it occurs at starting position i with *cumulative probability* $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) > 0$. Given a *cumulative weight threshold* $1/z \in (0,1]$, we say factor u is *z-valid*, if it occurs at position i with cumulative probability $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \ge 1/z$.

Example 4.2.2. Let $X = \texttt{ab[(a,0.5),(b,0.5)][(a,0.5),(b,0.5)]bab}$ and $1/z = 1/8$. String $u = \texttt{baaba}$ is a z-valid factor of X since u occurs at position 1 with cumulative probability $1/4 \ge 1/z = 1/8$.

Definition 4.2.3. Given a cumulative weight threshold $1/z \in (0,1]$, a weighted string X of length m is a *z-palindrome* if and only if there exists at least one z-valid factor u of X of length m which is a palindrome.

Example 4.2.3. Let $X = \texttt{a[(a,0.5),(b,0.5)]bab[(a,0.4),(b,0.6)]a}$ of length $m = 7$ and $1/z = 1/8$. $u = \texttt{abbabba}$ is a z-valid factor of X of length 7 and u is a palindrome. Hence we say X is a z-palindrome.

If the weighted string $X[i\,..\,j]$ is a z-palindrome, we analogously define the number $\frac{i+j}{2}$ as the center of $X[i\,..\,j]$ in X and $\frac{j-i+1}{2}$ as the radius of $X[i\,..\,j]$.

Definition 4.2.4. Let X be a weighted string of length n, $1/z \in (0,1]$ a cumulative weight threshold, and $X[i\,..\,j]$, where $0 \le i \le j \le n-1$, a z-palindrome. Then $X[i\,..\,j]$ is a *maximal z-palindrome* if there is no other z-palindrome in X with center $\frac{i+j}{2}$ and larger radius.

A maximal z-palindrome can thus also be encoded as a pair (c,r).

Definition 4.2.5. Let X be a weighted string of length n, $1/z \in (0,1]$ a cumulative weight threshold, and $X[i\,..\,j]$, where $0 \le i \le j \le n-1$, a z-palindrome. Then $X[i\,..\,j]$ is a *longest z-palindrome* if there is no other longer z-palindrome in X ending at position j.

We study the following computational problems.

SMALLEST MAXIMAL z-PALINDROMIC FACTORIZATION

Input: A weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$

Output: $X_1, X_2, \ldots, X_\ell$, if any, such that $X = X_1X_2\ldots X_\ell$, X_i, for all $1 \leq i \leq \ell$, is a maximal z-palindrome, and ℓ is minimal.

We call this output sequence X_1, X_2, $\ldots$, X_ℓ, i.e. when ℓ is minimal, a *smallest maximal z-palindromic factorization* of X.

Example 4.2.4. Given a weighted string

$$X = \texttt{a[(a,0.5),(b,0.5)]bacc[(a,0.3),(b,0.7)][(a,0.6),(b,0.4)]ba},$$

and a cumulative weight threshold $1/z = 1/5$, a smallest maximal z-palindromic factorization of X is

$$\texttt{a[(a,0.5),(b,0.5)]ba,c,c,[(a,0.3),(b,0.7)],[(a,0.6),(b,0.4)]ba}.$$

LONGEST z-PALINDROMIC ARRAY
Input: A weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$
Output: Longest z-palindromic array of X, $X[i\,..\,j]$ where $0 \le i \le j \le n-1$, such that $X[0]$, $X[i\,..\,1]$, $X[i\,..\,2]$, $X[i\,..\,3]$,..., $X[i\,..\,n-1]$, for $0 \le j \le n-1$, each $X[i\,..\,j]$ is a longest z-palindrome in X ending at position j.

Example 4.2.5. Given the weighted string $X =$

`a[(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]aa`

of length $n = 10$ and a cumulative weight threshold $1/z = 1/4$, a longest z-palindromic array of X is as follows:

$X[0] = a$,

$X[0\,..\,1] = a[(a,0.5),(b,0.5)]$,

$X[1\,..\,2] = [(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]$,

$X[1\,..\,3] = (a,0.5),(b,0.5)][(b,0.5),(c,0.5)]b$,

$X[0\,..\,4] = a[(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]ba$,

$X[3\,..\,5] = ba[(a,0.5),(b,0.5)]$,

$X[2\,..\,6] = [(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c$,

$X[5\,..\,7] = [(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]$,

$X[4\,..\,8] = a[(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]a$,

$X[7\,..\,9] = [(a,0.5),(c,0.5)]aa$.

4.2.2 Useful Properties of Maximal Palindromes

Fact 4.2.1 ([33])**.** Given a string x, $\mathcal{MP}(x)$ can be computed in time $\mathcal{O}(|x|)$.

Example 4.2.6. Given string $x = \texttt{abaab}$, we construct string $x\#x^R\$ = \texttt{abaab\#baaba\$}$. All maximal palindromes can be computed using the suffix tree of $x\#x^R\$$ (See Figure 4.2). Table 4.1 shows all odd-length palindromes and Table 4.2 shows all even-length palindromes, where $\mathsf{lcp}(i, j)$ denotes the longest common prefix of the suffixes starting at positions i and j of $x\#x^R\$$.

Index i	Index j	$\mathsf{lcp}(i, j)$	Palindrome	Center	Length
0	10	a	a	0	1
1	9	ba	aba	1	3
2	8	a	a	2	1
3	7	a	a	3	1
4	6	b	b	4	1

Table 4.1 Computing odd-length maximal palindromes of $x = \texttt{abaab}$ using the suffix tree of $x\#x^R\$ = \texttt{abaab\#baaba\$}$.

Index i	Index j	$\mathsf{lcp}(i,j)$	Palindrome	Center	Length
1	10	ε		0.5	0
2	9	ε		1.5	0
3	8	ab	baab	2.5	4
4	7	ε		3.5	0

Table 4.2 Computing even-length maximal palindromes of $x =$ abaab using the suffix tree of $x\#x^R\$ =$ abaab#baaba$.

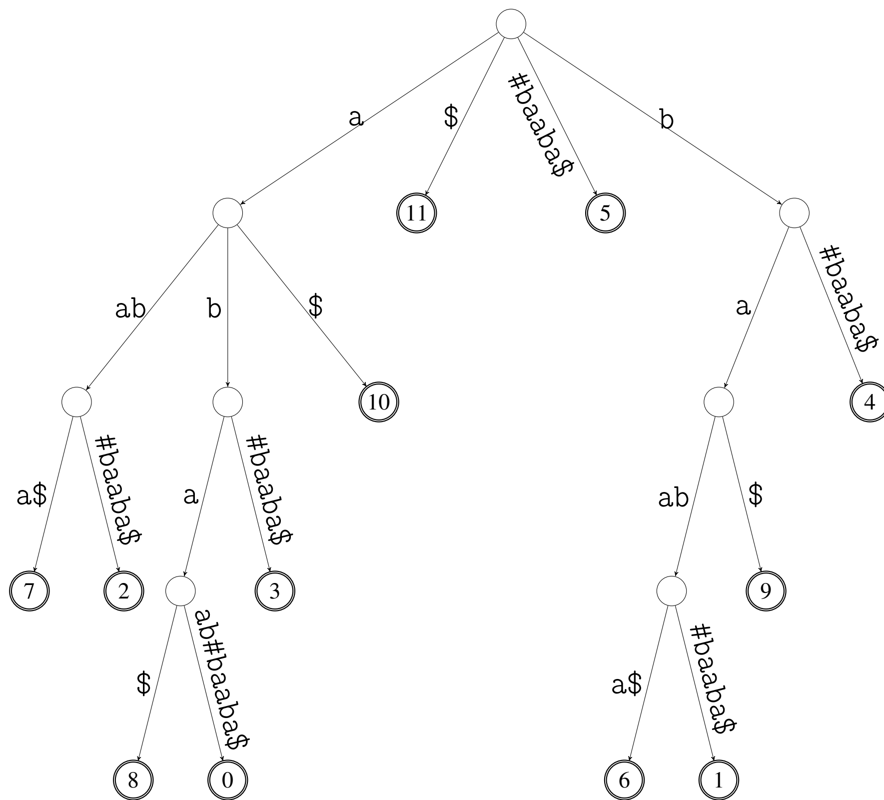


Fig. 4.2 Suffix tree of $x\#x^R\$ =$ abaab#baaba$; double-lined nodes represent terminal nodes labeled with the associated indices.

4.3 Algorithms

4.3.1 Computation of Smallest Maximal z-Palindromic Factorization

In this section, we present an algorithm to compute a smallest maximal z-palindromic factorization of a given weighted string X of length n for a given cumulative threshold $1/z \in (0,1]$. Our algorithm follows the one of Alatabbi *et al.* for computing a smallest maximal palindromic factorization of standard strings [2] with some crucial modifications.

Why the algorithm of Alatabbi *et al.* cannot be applied for weighted strings. Odd-length maximal palindromes centered at position i of a standard string x can be computed by finding the longest common prefix of suffixes $x[i\,..\,n-1]$ and $x^R[n-i-1\,..\,n-1]$. The longest common prefix of two suffixes can be found in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time pre-processing of the suffix tree of $x\#x^R\$$, where $\#, \$ \notin \Sigma$, using LCA queries; using a similar computation, we can find all even-length maximal palindromes [33].

The length of the longest common z-valid prefix of any two suffixes of our weighted string X can be computed in time $\mathcal{O}(z)$ after $\mathcal{O}(nz)$-time pre-processing using the suffix-tree-based Weighted Index (WI) of [13] (inspect also Figure 4.3). However, this does not guarantee that the two corresponding common z-valid prefixes shall form a maximal z-palindrome: the two prefixes are z-valid by definition of the WI but their concatenation that forms a palindrome *may not* be z-valid because its occurrence probability may drops below $1/z$.

We hence proceed as follows. By $\mathcal{MP}(X,z)$, we denote the set of center-distinct maximal z-palindromes of our weighted string X. Recall that we can represent a z-palindrome with center c and radius r by (c,r). For each position of X we define the *heaviest letter* as the letter with the maximum probability (breaking ties arbitrarily). We consider the string obtained from X by choosing at each position the heaviest letter. We call this the *heavy string* of X.

We define a collection $\mathcal{Z}_X$ of z SPECIAL-WEIGHTED STRINGS of X, denoted by $\mathcal{Z}_k$, $0 \leq k < z$. Each $\mathcal{Z}_k$ is of length n and it has the following properties. Each position j in $\mathcal{Z}_k$ contains at most one letter with positive probability and it corresponds to position j in X. If f is a z-valid factor occurring at position j of X, then f occurs at position j in some of the $\mathcal{Z}_k$'s. The combinatorial observation telling us that this is possible is due to Barton et al [12]. For clarity of presentation we write $\mathcal{Z}_k$'s as standard strings.

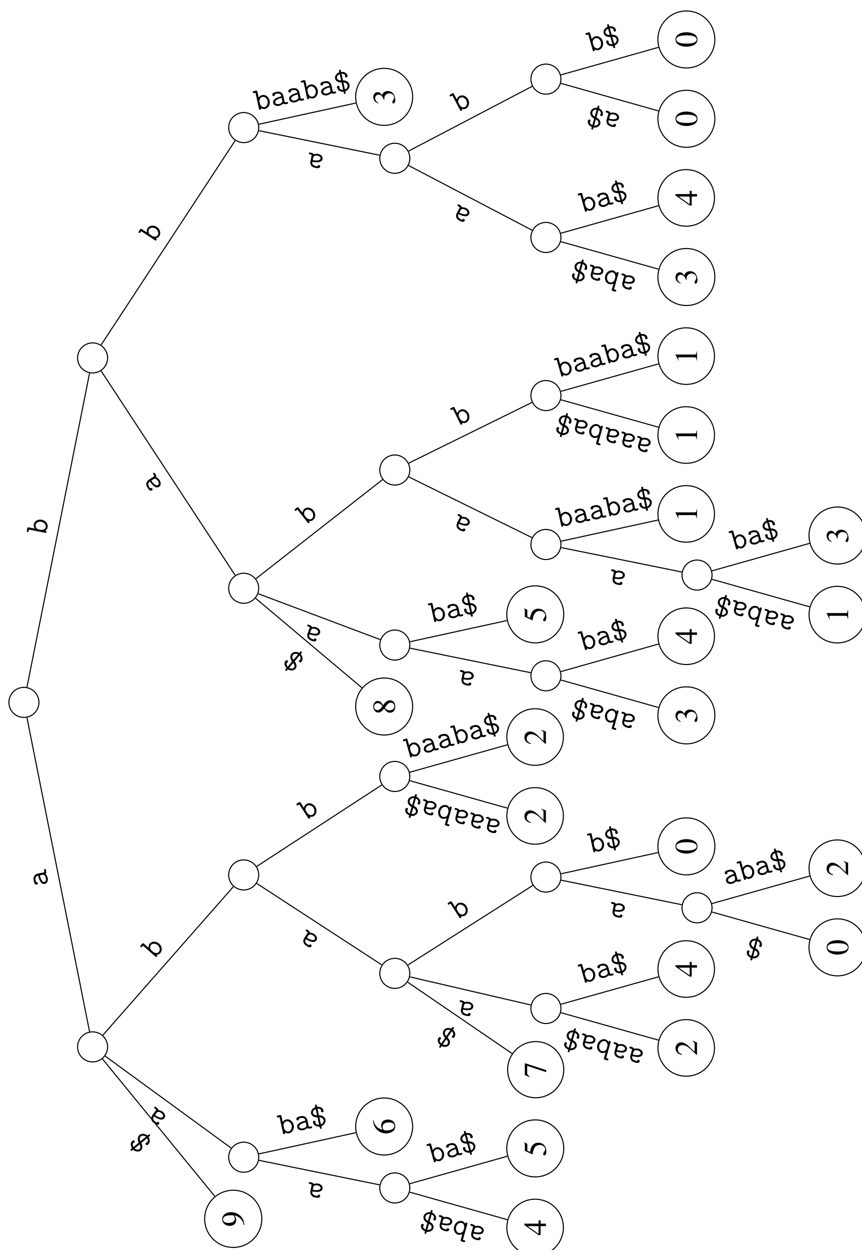


Fig. 4.3 The WI for X and $1/z$ shown in Example 4.3.1 (labels of edges to terminal nodes are appended with a letter $\$ \notin \Sigma$ for convenience).

Example 4.3.1. Given the weighted string

$$X = \texttt{[(a,0.5),(b,0.5)]bab[(a,0.5),(b,0.5)][(a,0.5),(b,0.5)]aaba}$$

and a cumulative weight threshold $1/z = 1/4$, we have:

$$\mathcal{Z}_X = \{\mathcal{Z}_0, \mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3\} = \{\texttt{ababaaaaba}, \texttt{ababbaaaba}, \texttt{bbababaaba}, \texttt{bbabbbaaba}\}.$$

Lemma 4.3.1 ([12]). Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, the z SPECIAL-WEIGHTED STRINGS of X can be constructed in time and space $\mathcal{O}(nz)$.

Proof. We first construct the WI of X in time and space $\mathcal{O}(nz)$ [13]. We begin from all the leaf nodes labelled with index 0 and initiate z/p strings for each node with a path-label having an occurrence probability $1/p$ (inspect also Figure 4.3).

1. Say we are considering leaf node u with index 0. We read the depth $\mathcal{D}(u)$ and the path-label $\mathcal{L}(u)$. We set $\mathcal{Z}_u[0\,..\,\mathcal{D}(u)-1] = \mathcal{L}(u)$.

2. We follow the suffix-link of the parent of u.

3. We find the node u' such that $\mathcal{L}(u') = \mathcal{Z}_u[1\,..\,\mathcal{D}(u)-1]$ by progressing down the edges in the tree, reading only the first letter of each edge, and keeping track of the current depth until the desired depth is reached.

4. Finally, we continue by spelling letters downwards the tree until we reach a leaf node v with the next index 1. At the same time, we append these letters to $\mathcal{Z}_u$. In

order to decide which path to take at explicit nodes for a certain $\mathcal{Z}_u$, we check that the occurrence probability of the factor starting at position 1 is at least $1/z$; and we have only $\sigma = \mathcal{O}(1)$ letters to check. This is easy to maintain by accessing the associated occurrence probabilities in X.

5. When we arrive at node v, we follow the suffix-link of the parent of v, and repeat Steps 3-5, until we have that $|\mathcal{Z}_u| = n$.

Let us analyse the time complexity of this procedure. It is clear that the total time required for Steps 1, 2, 4, and 5 for constructing one special-weighted string $\mathcal{Z}_u$ is bounded by $\mathcal{O}(n)$. Step 3 takes time proportional to the number of nodes we skip at each iteration; this is variable in each step but amortizes over the entire string giving us a total of $\mathcal{O}(n)$ for this step and therefore for the whole process. It suffices to note that the number of distinct leafs with the same index i in WI are at most z [13]. This is because there exist at most z right-maximal z-valid factors starting at any position i of X [4]. We thus obtain the result. □

Fact 4.3.2. Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, we have that $\mathcal{MP}(X,z) \subseteq \mathcal{MP}(\mathcal{Z}_0,z) \cup \mathcal{MP}(\mathcal{Z}_1,z) \cup \ldots \cup \mathcal{MP}(\mathcal{Z}_{z-1},z)$.

Proof. Suppose $U = X[i\,..\,j]$ is a maximal z-palindrome of center $c = \frac{i+j}{2}$ and radius $r = \frac{j-i+1}{2}$. By definition of U there must exist a z-valid palindromic factor u of U of radius r. Therefore, by definition of the SPECIAL-WEIGHTED STRINGS of X, u must be a z-valid factor of some $\mathcal{Z}_k$ and thus $(c,r) \in \mathcal{MP}(\mathcal{Z}_k,z)$. □

There are two steps for the correct computation of $\mathcal{MP}(X,z)$. First, we compute the set $\mathcal{A}_k$ of all maximal palindromes of the heavy string of $\mathcal{Z}_k$, for all $0 \le k < z$, using Fact 4.2.1. We then need to adjust the radius of each reported palindrome for $\mathcal{Z}_k$ to ensure that it is z-valid in X (the center should not change). To achieve this, we compute an array $\mathcal{R}_k$, for each $\mathcal{Z}_k$, such that $\mathcal{R}_k[2c]$ stores the radius of the longest factor at center c in $\mathcal{Z}_k$ which is a z-valid factor of X at center c, e.g. $\mathcal{R}_k[2c] = \frac{j-i+1}{2}$, $c = (i+j)/2$, if $\mathcal{Z}_k[i\,..\,j]$ is a z-valid factor of X centered at c, and $\mathcal{Z}_k[i-1\,..\,j+1]$ is not a z-valid factor of X. By Fact 4.3.2, we cannot guarantee that all (c,r) in $\mathcal{MP}(\mathcal{Z}_k,z)$ are necessarily in $\mathcal{MP}(X,z)$. Hence, the second step is to compute $\mathcal{MP}(X,z)$ from $\mathcal{MP}(\mathcal{Z}_k,z)$ by taking the maximum radius per center and filtering out everything else.

Lemma 4.3.3. Given a weighted string X of length n, a cumulative weight threshold $1/z \in (0,1]$, and the SPECIAL-WEIGHTED STRINGS $\mathcal{Z}_X$ of X, each $\mathcal{R}_k$, $0 \le k < z$, can be computed in time $\mathcal{O}(n)$.

Proof. By $< i,c,j >$, where $0 \le i \le c \le j \le n-1$, we denote a factor of $\mathcal{Z}_k$ that has starting position i, ending position j and center $c = (i+j)/2$. We further denote the occurrence probability of $< i,c,j >$ in $\mathcal{Z}_k$ by $\Pi_{<i,c,j>} = \prod_{q=i}^{j} \pi_q(\mathcal{Z}_k[q])$. A factor $< i,c,j >$ of $\mathcal{Z}_k$ is called a special maximal z-valid factor of $\mathcal{Z}_k$ if $\Pi_{<i,c,j>} \ge 1/z$ and $\Pi_{<i-1,c,j+1>} < 1/z$.

For each $\mathcal{Z}_k$, we compute $\mathcal{R}_k$ from left to right. If we have $\Pi_{<0,0,0>} \ge 1/z$, we set $\mathcal{R}_k[0] = \frac{1}{2}$. If not, we go to the next position until we find a valid letter, say at position ℓ; then we have $\mathcal{R}_k[0] = \cdots = \mathcal{R}_k[2\ell-1] = 0$ and $\mathcal{R}_k[2\ell] = \frac{1}{2}$. Note that this corresponds to the first special maximal z-valid factor. Suppose we have a special maximal z-valid

factor $< i,c,j >$ and $\mathcal{R}_k[2c] = \frac{j-i+1}{2}$, we show how to compute $\mathcal{R}_k[2c+1]$, which is the length of the special maximal z-valid factor at center $c' = \frac{2c+1}{2}$. We add the letter after $< i,c,j >$, so we have $< i,c',j+1 >$. We compute $\Pi_{<i,c',j+1>}$, which is simply $\Pi_{<i,c,j>} \times \pi_{j+1}(\mathcal{Z}_k[j+1])$. If $\Pi_{<i,c',j+1>} \geq 1/z$, the special maximal z-valid factor at center c' should be $< i,c',j+1 >$ and $\mathcal{R}_k[2c+1] = \mathcal{R}_k[2c] + \frac{1}{2} = \frac{j-i+2}{2}$. Factor $< i-1,c',j+2 >$ cannot be z-valid, since if $\Pi_{<i-1,c',j+2>} \geq 1/z$, we must have $\Pi_{<i-1,c,j+1>} \geq \Pi_{<i-1,c',j+2>} \geq 1/z$, which gives a longest special maximal z-valid at center c, namely $< i-1,c,j+1 >$, a contradiction. For $\Pi_{<i,c',j+1>} < 1/z$, the special maximal z-valid factor at center c' is $< i+1,c',j >$ since it always holds that $\Pi_{<i+1,c',j>} \geq \Pi_{<i,c,j>} \geq 1/z$. Therefore $\mathcal{R}_k[2c+1] = \mathcal{R}_k[2c] - \frac{1}{2} = \frac{j-i}{2}$.

Each center needs only to be considered once and there exist $2n-1$ distinct centers in each $\mathcal{Z}_k$. Therefore each $\mathcal{R}_k$ can be computed in $\mathcal{O}(n)$ time. □

Fact 4.3.4 (Trivial)**.** Let $x[i\,..\,j]$ be a palindrome of string x with center c and let u, $|u| < j-i+1$, be a factor of x with center c. Then u is also a palindrome.

After computing $\mathcal{A}_k$ and $\mathcal{R}_k$, we perform the following check for each palindrome $(c,r) \in \mathcal{A}_k$. If $r > \mathcal{R}_k[2c]$, the palindrome with radius r is not z-valid but the factor with radius $\mathcal{R}_k[2c]$ is z-valid and maximal (by definition) and palindromic (by Fact 4.3.4); if $r \leq \mathcal{R}_k[2c]$, the palindrome with radius r_i must be z-valid and it is maximal. Therefore we set $(c,r) \in \mathcal{MP}(\mathcal{Z}_k,z)$, such that $r = \min\{r,\mathcal{R}_k[2c]\}$, $0 \leq 2c \leq 2n-2$, and $r \geq 1/2$.

To go from $\mathcal{MP}(\mathcal{Z}_k, z)$ to $\mathcal{MP}(X, z)$ we need to take the maximum radius for each center. Therefore for each center $c/2$, $0 \leq c \leq 2n-2$, we set $(c/2, r) \in \mathcal{MP}(X, z)$, such that $r = \max\{r_k | (c/2, r_k) \in \mathcal{MP}(\mathcal{Z}_k, z), 0 \leq k < z\}$.

Theorem 4.3.5. Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0, 1]$, all maximal z-palindromes in X can be computed in time and space $\mathcal{O}(nz)$.

After the computation of $\mathcal{MP}(X, z)$, we are in a position to apply the algorithm by Alatabbi et al [2] to find a smallest maximal z-palindromic factorization. We define a list $\mathcal{F}$ such that $\mathcal{F}[i]$, $0 \leq i \leq n-1$, stores the set of the lengths of all maximal z-palindromes ending at position i in X. We also define a list $\mathcal{U}$ such that $\mathcal{U}[i]$, $0 \leq i \leq n-1$, stores the set of positions j, such that $j+1$ is the starting position of a maximal z-palindrome in X and i is the ending position of this z-palindrome. Thus for a given $\mathcal{F}[i] = \{\ell_0, \ell_1, \ldots, \ell_q\}$, we have that $\mathcal{U}[i] = \{i-\ell_0, i-\ell_1, \ldots, i-\ell_q\}$. Note that $\mathcal{U}[i]$ can contain a "-1" element if there exists a maximal z-palindrome starting at position 0 and ending at position i. Note that the number of elements in $\mathcal{MP}(X, z)$ is at most $2n-1$, and, hence, $\mathcal{F}$ and $\mathcal{U}$ can contain at most $2n-1$ elements. The lists $\mathcal{F}$ and $\mathcal{U}$ can be computed trivially from $\mathcal{MP}(X, z)$.

Finally, we define a directed graph $\mathcal{G}_X = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{i \mid -1 \leq i \leq n-1\}$ and $\mathcal{E} = \{(i, j) \mid j \in \mathcal{U}[i]\}$. Note that (i, j) is a directed edge from i to j. We do a breath first search on $\mathcal{G}_X$ assuming the vertex $n-1$ as the source and identify the shortest path from $n-1$ to -1, which gives a factorization.

We formally present the above as Algorithm SMPF for computing a smallest maximal z-palindromic factorization and obtain the following result.

Theorem 4.3.6. Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, Algorithm SMPF correctly solves the problem SMALLEST MAXIMAL z-PALINDROMIC FACTORIZATION in time and space $\mathcal{O}(nz)$.

Proof. The correctness follows from Theorem 4.3.5 for computing $\mathcal{MP}(X,z)$ and from the correctness of the algorithm in [2] for computing a smallest maximal palindromic factorization.

By Lemma 4.3.1, the construction of the SPECIAL-WEIGHTED STRINGS can be done in time and space $\mathcal{O}(nz)$. Computing $\mathcal{A}_k$ and $\mathcal{R}_k$, for all $0 \leq k < z$, can be done in total time $\mathcal{O}(nz)$ by Fact 4.2.1 and Lemma 4.3.3, respectively. From there on, computing $\mathcal{MP}(X,z)$ can be done in time $\mathcal{O}(nz)$. The lists $\mathcal{F}$ and $\mathcal{U}$ can be computed in time $\mathcal{O}(n)$ since the size of $\mathcal{MP}(X,z)$ is no more than $2n-1$. There exist in total $n+1$ vertices in $\mathcal{G}_X$. The number of edges $\mathcal{E}$ depends on $\mathcal{U}$, which contains no more than $2n$ elements; we have $\mathcal{E} = \mathcal{O}(n)$. Therefore, the construction of $\mathcal{G}_X$ and the breadth first search can be done in time $\mathcal{O}(\mathcal{V}+\mathcal{E}) = \mathcal{O}(n)$. The identification of the desired path can also be done easily if we do some simple bookkeeping during the breadth first search. The total running time of Algorithm SMPF is thus $\mathcal{O}(nz)$ and the space required is $\mathcal{O}(nz)$. □

1 **Algorithm** *SMPF*$(X, n, 1/z)$
2 Construct the set $\mathcal{Z}_X$ of SPECIAL-WEIGHTED STRINGS of X;
3 **foreach** $\mathcal{Z}_k \in \mathcal{Z}_X$ **do**
4 $\mathcal{A}_k \leftarrow$ maximal palindromes of the heavy string of $\mathcal{Z}_k$;
5 Compute $\mathcal{R}_k$ for $\mathcal{Z}_k$;
6 $\mathcal{MP}(\mathcal{Z}_k, z) \leftarrow$ EMPTYLIST();
7 **foreach** $(c, r) \in \mathcal{A}_k$ **do**
8 $r \leftarrow \min\{r, \mathcal{R}_k[2c]\}$;
9 **if** $r \geq \frac{1}{2}$ Insert (c, r) in $\mathcal{MP}(\mathcal{Z}_k, z)$;
10 $\mathcal{MP}(X, z) \leftarrow$ EMPTYLIST();
11 **foreach** $c \in [0, 2n-2]$ **do**
12 $r \leftarrow \max\{r_k | (c/2, r_k) \in \mathcal{MP}(\mathcal{Z}_k, z), 0 \leq k < \lfloor z \rfloor\}$;
13 Insert $(c/2, r)$ in $\mathcal{MP}(X, z)$;
14 $\mathcal{F} \leftarrow$ EMPTYLIST();
15 $\mathcal{U} \leftarrow$ EMPTYLIST();
16 **foreach** $(c, r) \in \mathcal{MP}(X, z)$ **do**
17 $j \leftarrow \lfloor c + r \rfloor$;
18 Insert $2r$ in $\mathcal{F}[j]$;
19 Insert $c - r$ in $\mathcal{U}[j]$;
20 Construct directed graph $\mathcal{G}_X = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{i \mid -1 \leq i \leq n-1\}$, $\mathcal{E} = \{(i, j) \mid j \in \mathcal{U}[i]\}$ and (i, j) is a directed edge from i to j;
21 Breadth first search on $\mathcal{G}_X$ assuming the vertex $n-1$ as the source;
22 Identify the shortest path $P \equiv \langle n-1 = p_\ell, p_{\ell-1}, \ldots, p_2, p_1, p_0 = -1 \rangle$;
23 **Return** $X[0\,..\,p_1], X[p_1+1\,..\,p_2], \ldots, X[p_{\ell-1}+1\,..\,p_\ell]$;

Example 4.3.2. Given the weighted string $X =$

`a[(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]aa`

of length $n = 10$ and a cumulative weight threshold $1/z = 1/4$, we proceed as follows. First, we construct the SPECIAL-WEIGHTED STRINGS of X: $\mathcal{Z}_X = \{\mathcal{Z}_0, \mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3\} =$ $\{$`aabbaacaaa,aacbaaccaa,abbbabcaaa,abcbabccaa`$\}$ (see Appendix A for the construction using the WI). Second, we compute $\mathcal{A}_k$ and $\mathcal{R}_k$, and then we compute $\mathcal{MP}(\mathcal{Z}_k, z)$, $0 \leq k < \lfloor z \rfloor$. From $\mathcal{MP}(\mathcal{Z}_k, z)$, we compute $\mathcal{MP}(X, z)$, $\mathcal{F}$, and $\mathcal{U}$ (see Table 4.3). Finally, the graph $\mathcal{G}_X$ is constructed from $\mathcal{U}$; and the shortest path $P = \langle 9, 6, 1, -1 \rangle$ is found (see Fig. 4.4; corresponding edges are shown as dashed edges).

The output sequence is thus $X[0\,..\,1], X[2\,..\,6], X[7\,..\,9] =$ `a[(a,0.5),(b,0.5)]`, `[(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c`, `[(a,0.5),(c,0.5)]aa`.

Index	$\mathcal{MP}(X,z)$	j	$\mathcal{F}[j]$	$\mathcal{U}[j]$
0	(0, 0.5)	0	$\{1\}$	$\{-1\}$
1	(0.5, 1)			
2	(1, 0.5)	1	$\{1,2\}$	$\{0,-1\}$
3	(1.5, 1)			
4	(2, 2.5)	2	$\{2\}$	$\{0\}$
5	(2.5, 2)			
6	(3, 0.5)	3	$\{0,1\}$	$\{3,2\}$
7	(3.5, 0)			
8	(4, 2.5)	4	$\{4,5\}$	$\{0,-1\}$
9	(4.5, 1)			
10	(5, 0.5)	5	$\{0,1,2\}$	$\{5,4,3\}$
11	(5.5, 0)			
12	(6, 2.5)	6	$\{5\}$	$\{1\}$
13	(6.5, 3)			
14	(7, 0.5)	7	$\{1\}$	$\{6\}$
15	(7.5, 1)			
16	(8, 1.5)	8	$\{2,5\}$	$\{6,3\}$
17	(8.5, 1)			
18	(9, 0.5)	9	$\{1,2,3\}$	$\{8,7,6\}$

Table 4.3 Computing $\mathcal{F}$ and $\mathcal{U}$ from $\mathcal{MP}(X,z)$ for X and $1/z$ shown in Example 4.3.2.

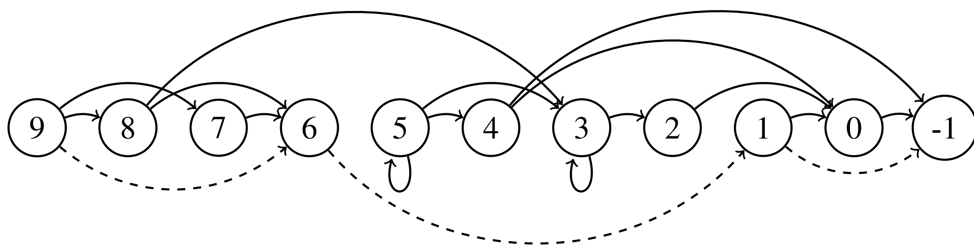


Fig. 4.4 The graph $\mathcal{G}_X$ for X and $1/z$ shown in Example 4.3.2.

4.3.2 Computation of Longest z-Palindromic Array

By Theorem 4.3.5, given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, all maximal z-palindromes in X can be computed in time and space $\mathcal{O}(nz)$. Now we present the Algorithm LPA for computing longest z-palindromic array. We also define a list $\mathcal{PA}$ such that $\mathcal{PA}[j]$, $0 \leq j \leq n-1$, stores the position $i^{'}$, such that $i^{'}+1$ (or $\mathcal{PA}[j]+1$) is the starting position of a longest z-palindrome in X and j is the ending position of this z-palindrome. Note that $\mathcal{PA}[j]$ is either a position which holds a maximal palindrome starting at position $i^{'}+1$ and ending at position j, or a position $\mathcal{PA}[j+1]+1$ which presents the longest sub-palindrome of a palindrome ending at position $j+1$ with the same center. As wll known, if $X[i-1\,..\,j+1]$, for $j-i>0$, is a palindrome, and then $X[i\,..\,j]$ must be a palindrome.

Theorem 4.3.7. Given a weighted string X of length n and a cumulative weight threshold $1/z \in (0,1]$, Algorithm LPA correctly solves the problem LONGEST z-PALINDROMIC ARRAY in time and space $\mathcal{O}(nz)$.

1 **Algorithm** $LPA(X,n,1/z)$
2 Construct the set $\mathcal{Z}_X$ of SPECIAL-WEIGHTED STRINGS of X;
3 **foreach** $\mathcal{Z}_k \in \mathcal{Z}_X$ **do**
4 $\mathcal{A}_k \leftarrow$ maximal palindromes of the heavy string of $\mathcal{Z}_k$;
5 Compute $\mathcal{R}_k$ for $\mathcal{Z}_k$;
6 $\mathcal{MP}(\mathcal{Z}_k,z) \leftarrow \text{EMPTYLIST}()$;
7 **foreach** $(c,r) \in \mathcal{A}_k$ **do**
8 $r \leftarrow \min\{r, \mathcal{R}_k[2c]\}$;
9 **if** $r \geq \frac{1}{2}$ Insert (c,r) in $\mathcal{MP}(\mathcal{Z}_k,z)$;
10 $\mathcal{MP}(X,z) \leftarrow \text{EMPTYLIST}()$;
11 **foreach** $c \in [0,2n-2]$ **do**
12 $r \leftarrow \max\{r_k | (c/2, r_k) \in \mathcal{MP}(\mathcal{Z}_k,z), 0 \leq k < \lfloor z \rfloor\}$;
13 Insert $(c/2,r)$ in $\mathcal{MP}(X,z)$;
14 $\mathcal{F} \leftarrow \text{EMPTYLIST}()$;
15 $\mathcal{U} \leftarrow \text{EMPTYLIST}()$;
16 $\mathcal{PA} \leftarrow \text{EMPTYLIST}()$;
17 **foreach** $(c,r) \in \mathcal{MP}(X,z)$ **do**
18 $j \leftarrow \lfloor c+r \rfloor$;
19 Insert $2r$ in $\mathcal{F}[j]$;
20 Insert $c-r$ in $\mathcal{U}[j]$;
21 $\mathcal{PA}[n-1] \leftarrow \min\{\mathcal{U}[n-1]\}$;
22 **foreach** $j \in [n-2,0]$ **do**
23 $\mathcal{PA}[j] \leftarrow \min\{\mathcal{PA}[j+1]+1, \mathcal{U}[j]\}$;
24 **Return** $X[0], X[(\mathcal{PA}[1]+1)\,..\,1], X[(\mathcal{PA}[2]+1)\,..\,2], X[(\mathcal{PA}[3]+1)\,..\,3], .., X[(\mathcal{PA}[n-1]+1)\,..\,n-1]$;

Example 4.3.3. Given the weighted string $X =$

`a[(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]aa`

of length $n = 10$ and a cumulative weight threshold $1/z = 1/4$, we proceed as follows. First, we construct the SPECIAL-WEIGHTED STRINGS of X: $\mathcal{Z}_X = \{\mathcal{Z}_0, \mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3\} =$ $\{$`aabbaacaaa`, `aacbaaccaa`, `abbbabcaaa`, `abcbabccaa`$\}$ (see Appendix A for the construction using the WI). Second, we compute $\mathcal{A}_k$ and $\mathcal{R}_k$, and then we compute $\mathcal{MP}(\mathcal{Z}_k, z)$, $0 \leq k < \lfloor z \rfloor$. From $\mathcal{MP}(\mathcal{Z}_k, z)$, we compute $\mathcal{MP}(X, z)$, $\mathcal{F}$, $\mathcal{U}$, and $\mathcal{PA}$ (see Table 4.4). Finally, the longest z-Palindromic array of X is found (see Fig. 4.5).

The output sequence is thus:

$X[0] = a$,

$X[0\,..\,1] = a[(a,0.5),(b,0.5)]$,

$X[1\,..\,2] = [(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]$,

$X[1\,..\,3] = (a,0.5),(b,0.5)][(b,0.5),(c,0.5)]b$,

$X[0\,..\,4] = a[(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]ba$,

$X[3\,..\,5] = ba[(a,0.5),(b,0.5)]$,

$X[2\,..\,6] = [(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c$,

$X[5\,..\,7] = [(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]$,

$X[4\,..\,8] = a[(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]a$,

$X[7\,..\,9] = [(a,0.5),(c,0.5)]aa$.

Index	$\mathcal{MP}(X,z)$	j	$\mathcal{F}[j]$	$\mathcal{U}[j]$	$\mathcal{PA}[j]$
0	(0, 0.5)	0	$\{1\}$	$\{-1\}$	-1
1	(0.5, 1)				
2	(1, 0.5)	1	$\{1,2\}$	$\{0,-1\}$	-1
3	(1.5, 1)				
4	(2, 2.5)	2	$\{2\}$	$\{0\}$	0
5	(2.5, 2)				
6	(3, 0.5)	3	$\{0,1\}$	$\{3,2\}$	0
7	(3.5, 0)				
8	(4, 2.5)	4	$\{4,5\}$	$\{0,-1\}$	-1
9	(4.5, 1)				
10	(5, 0.5)	5	$\{0,1,2\}$	$\{5,4,3\}$	2
11	(5.5, 0)				
12	(6, 2.5)	6	$\{5\}$	$\{1\}$	1
13	(6.5, 3)				
14	(7, 0.5)	7	$\{1\}$	$\{6\}$	4
15	(7.5, 1)				
16	(8, 1.5)	8	$\{2,5\}$	$\{6,3\}$	3
17	(8.5, 1)				
18	(9, 0.5)	9	$\{1,2,3\}$	$\{8,7,6\}$	6

Table 4.4 Computing $\mathcal{F}$, $\mathcal{U}$ and $\mathcal{PA}$ from $\mathcal{MP}(X,z)$ for X and $1/z$ shown in Example 4.3.3.

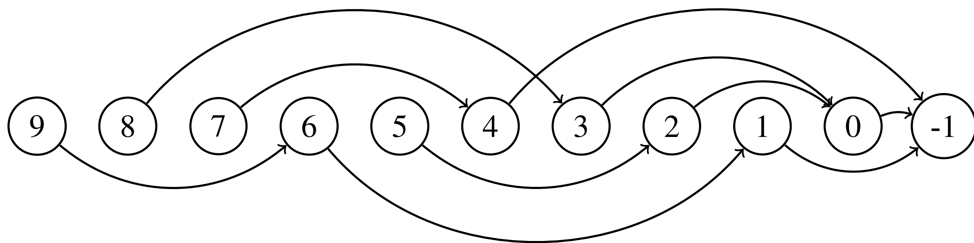


Fig. 4.5 $\mathcal{PA}$ for X and $1/z$, this graph shown in Example 4.3.3.

4.4 Implementation and Experiments

The program was implemented in the `C++` programming language and developed under GNU/Linux operating system. The input parameters are a (Multi)FASTA file with the input sequence(s), an integer $z > 1$. The output is a file with the set of maximal palindromes per input sequence. The experiments were conducted on a Desktop PC using one core of Intel Core i5-4690 CPU at 3.50GHz under GNU/Linux. The program was compiled with g++ version 4.8.4 at optimisation level 3 (-O3).

4.4.1 Smallest Maximal z-Palindromic Factorization

Algorithm SMALLEST MAXIMAL Z-PALINDROMIC FACTORIZATION was implemented as a program to compute the smallest maximal z-palindromic factorization in one or more input sequences.

Experiment 1. In the first experiment, our task was to establish the fact that the elapsed time and memory usage of the program grow linearly with z. As input datasets, for this experiment, we used synthetic DNA sequence of length 1MB. For this sequence we used different values of z. The results, for elapsed time and maximal memory usage, are plotted in Fig. 4.6. It becomes evident from the results that the elapsed time and memory usage of the program grow linearly with z.

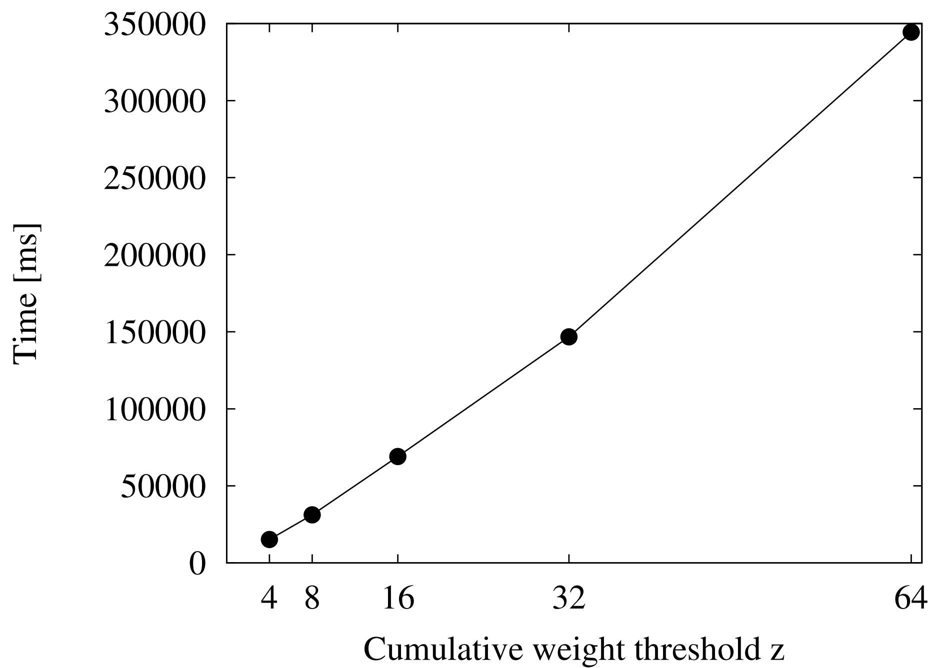
Experiment 2. In the second experiment, our task was to establish the fact that the elapsed time and memory usage of the program grow linearly with n, the length of the input sequence. As input datasets, for this experiment, we used synthetic DNA sequences ranging from 250KB to 4000KB. For each sequence we used constant values for $z = 8$. The results, for elapsed time and maximal memory usage, are plotted in

Fig. 4.7. It becomes evident from the results that the elapsed time and memory usage of the program grow linearly with n.

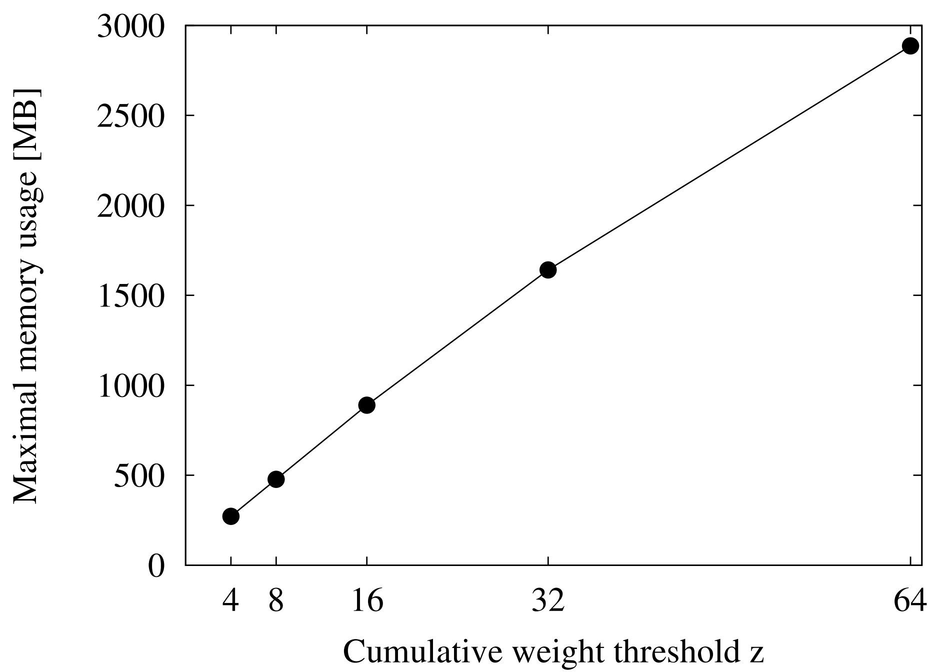
4.4.2 Longest z-Palindromic Array

Algorithm LONGEST Z-PALINDROMIC ARRAY was implemented as a program to compute the longest z-palindromic Array in one or more input sequences.

Experiment 3. In this experiment, our task was to establish the fact that the elapsed time and memory usage of the program grow linearly with n, the length of the input sequence. As input datasets, for this experiment, we used synthetic DNA ranging from 250KB to 4000KB. For each sequence we used constant values for $z = 8$. The results, for elapsed time and maximal memory usage, are plotted in Fig. 4.8. It becomes evident from the results that the elapsed time and memory usage of the program grow linearly with n.

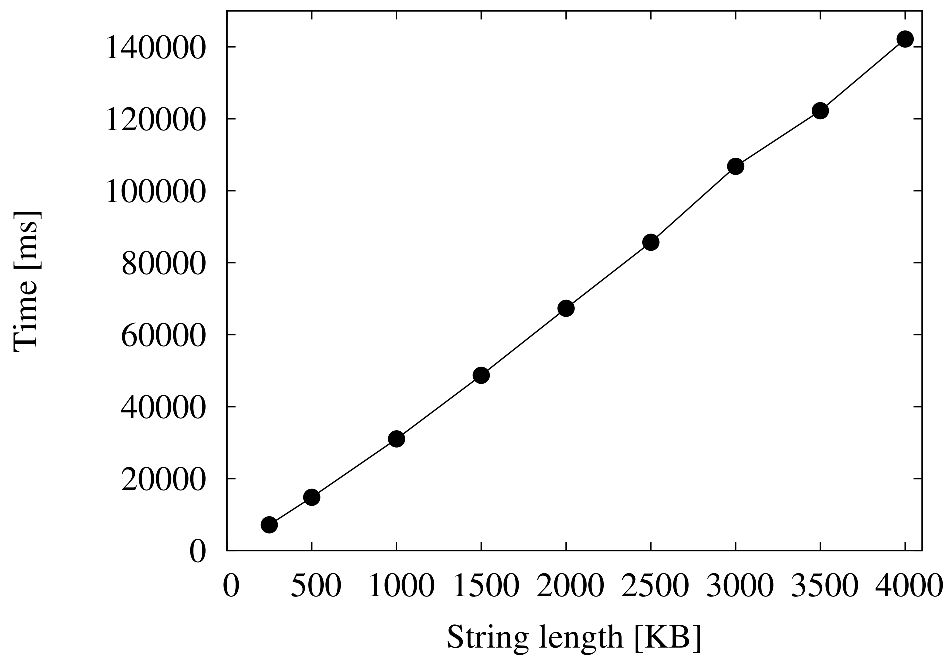


(a) Time for $n = 1$MB

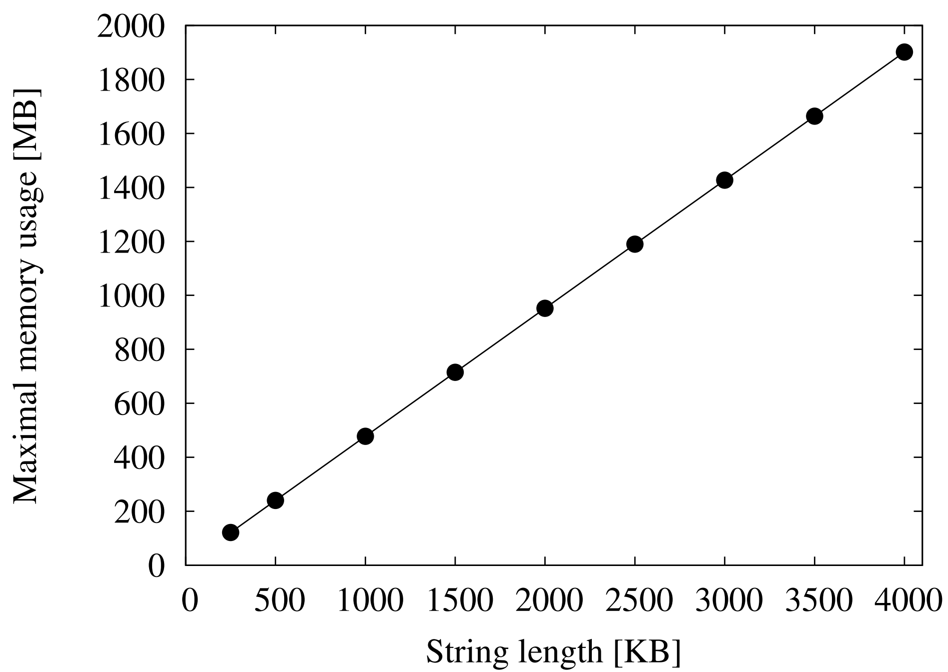


(b) Memory for $n = 1$MB

Fig. 4.6 Experiment 1. Elapsed time and maximal memory usage of Algorithm SMALLEST MAXIMAL Z-PALINDROMIC FACTORIZATION *using synthetic DNA ($\sigma = 4$) data of length* 1*MB for variable z.*

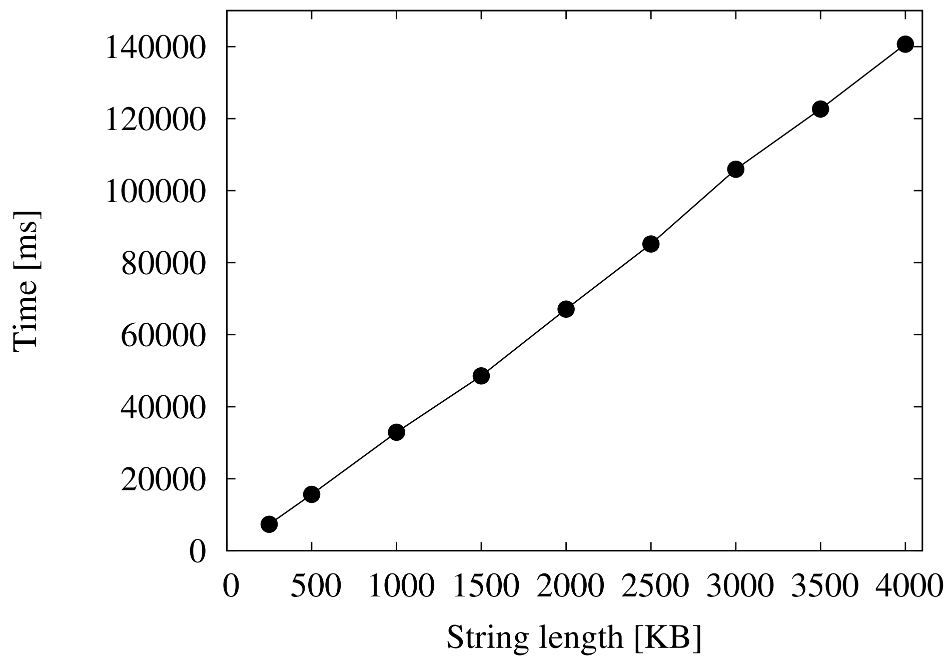


(a) Time for $z = 8$

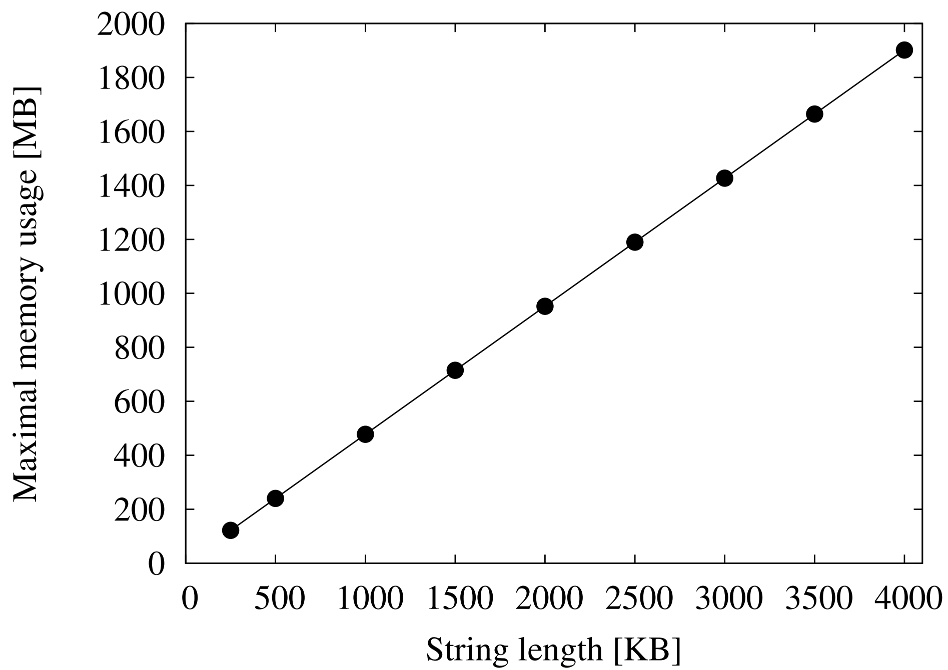


(b) Memory for $z = 8$

Fig. 4.7 Experiment 2. Elapsed time and maximal memory usage of Algorithm SMALLEST MAXIMAL Z-PALINDROMIC FACTORIZATION *using synthetic DNA ($\sigma = 4$) data of length* 250*KB to* 4000*KB.*



(a) Time for $z = 8$



(b) Memory for $z = 8$

Fig. 4.8 Experiment 3. Elapsed time and maximal memory usage of Algorithm LONGEST Z-PALINDROMIC ARRAY *using synthetic DNA ($\sigma = 4$) data of length* 250*KB to* 4000*KB.*

4.5 Conclusion

In this chapter, we presented an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm, where n is the length of the weighted string and $1/z$ is the given threshold, to compute a smallest maximal z-palindromic factorization of a weighted string. Moreover, we provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute all maximal z-palindromes in weighted strings.

We also presented an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute a longest z-palindromes array in weighted strings.

Finally, we made available an implementation of our algorithm, using synthetic data, show its efficiency in biological sequence analysis.

Chapter 5

Conclusions and Future work

In this thesis, we focused on computing certain structures in biological sequences using different algorithmic methods. Firstly, we addressed the problem of the computation of avoided words and overabundant words in biological sequences. One of the major contributions of our work is to present some algorithms that can be used effectively for computing such words, for instance, we presented an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm to compute all ρ-avoided words of length k in a given sequence of length n over a fixed-sized alphabet, we also presented an $\mathcal{O}(n)$-time and $\mathcal{O}(n)$-space algorithm to compute all overabundant words in a sequence x of length n over an integer alphabet. Furthermore, experimental results, using both real and synthetic data, which further highlight the effectiveness of this model, show the efficiency and applicability of our implementation in biological sequence analysis.

Secondly, we considered a special type of uncertain sequence called weighted string. One of the primary contributions of our work is to provide an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm, where n is the length of the weighted string and $1/z$ is the given

threshold, to compute a smallest maximal z-palindromic factorization of a weighted string. Moreover, we provided an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute all maximal z-palindromes in weighted strings. And then, we provided an $\mathcal{O}(nz)$-time and $\mathcal{O}(nz)$-space algorithm to compute a longest z-palindromes array in weighted strings. Last but not least, we made available an implementation of our algorithms, using synthetic data, show the efficiency of our implementation.

Many experiments have been left for the future due to lack of proper real data. One of the principal future work is to find the proper real data in weighted strings, and then to concern the analysis of the computation of all maximal z-palindromes in weighted strings, and also to concern the analysis of the longest z-palindromes array computation in weighted strings.

References

[1] Acquisti, C., Poste, G., Curtiss, D., and Kumar, S. (2007). Nullomers: really a matter of natural selection? *PLoS One*, 2(10):e1022.

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

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

[4] Amir, A., Chencinski, E., Iliopoulos, C., Kopelowitz, T., and Zhang, H. (2008). Property matching and weighted matching. *Theoretical Computer Science*, 395(2-3):298–310.

[5] Amir, A., Gotthilf, Z., and Shalom, B. R. (2010). Weighted LCS. *Journal of Discrete Algorithms*, 8(3):273–281.

[6] Apostolico, A., Bock, M. E., and Lonardi, S. (2003). Monotony of surprise and large-scale quest for unusual words. *Journal of Computational Biology*, 10(3-4):283–311.

[7] Apostolico, A., Bock, M. E., Lonardi, S., and Xu, X. (2000). Efficient detection of unusual words. *Journal of Computational Biology*, 7(1-2):71–94.

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

[9] Apostolico, A., Gong, F.-C., and Lonardi, S. (2004). Verbumculus and the discovery of unusual words. *Journal of Computer Science and Technology*, 19(1):22–41.

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

[11] Barton, C., Iliopoulos, C. S., and Pissis, S. P. (2014b). Optimal computation of all tandem repeats in a weighted sequence. *Algorithms for Molecular Biology*, 9(21).

[12] Barton, C., Kociumaka, T., Liu, C., Pissis, S. P., and Radoszewski, J. (2017). Indexing Weighted Sequences: Neat and Efficient. *CoRR*, abs/1704.07625.

[13] Barton, C., Kociumaka, T., Pissis, S. P., and Radoszewski, J. (2016a). Efficient Index for Weighted Sequences. In *Combinatorial Pattern Matching*, volume 54 of *LIPIcs*, pages 4:1–4:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Barton, C., Liu, C., and Pissis, S. P. (2016b). Linear-time computation of prefix table for weighted strings & applications. *Theoretical Computer Science*, 656:160–172.

[15] Barton, C., Liu, C., and Pissis, S. P. (2016c). On-line pattern matching on uncertain sequences and applications. In *International Conference on Combinatorial*

Optimization and Applications, volume 10043 of *LNCS*, pages 547–562. Springer International Publishing.

[16] Barton, C. and Pissis, S. P. (2017). Crochemore's partitioning on weighted strings and applications. *Algorithmica*, 80(2):496–514.

[17] Belazzougui, D. and Cunial, F. (2015). Space-efficient detection of unusual words. In *International Symposium on String Processing and Information Retrieval*, volume 9309 of *LNCS*, pages 222–233. Springer.

[18] Bender, M. A. and Farach-Colton, M. (2000). The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, volume 1776 of *LNCS*, pages 88–94. Springer-Verlag.

[19] Brendel, V., Beckmann, J. S., and Trifonov, E. N. (1986). Linguistics of nucleotide sequences: morphology and comparison of vocabularies. *Journal of Biomolecular Structure and Dynamics*, 4(1):11–21.

[20] Burge, C., Campbello, A. M., and Karlin, S. (1992). Over- and under-representation of short oligonucleotides in DNA sequences. *Proceedings of the National Academy of Sciences*, 89(4):1358–1362.

[21] Charalampopoulos, P., Crochemore, M., Fici, G., Mercaş, R., and Pissis, S. P. (2018). Alignment-free sequence comparison using absent words. *Information and Computation*.

[22] Crochemore, M., Hancart, C., and Lecroq, T. (2007). *Algorithms on strings*. Cambridge University Press.

[23] Cygan, M., Kubica, M., Radoszewski, J., Rytter, W., and Walen, T. (2016). Polynomial-time approximation algorithms for weighted LCS problem. *Discrete Applied Mathematics*, 204:38–48.

[24] Denise, A., Régnier, M., and Vandenbogaert, M. (2001). Assessing the statistical significance of overrepresented oligonucleotides. In *WABI*, volume 2149 of *LNCS*, pages 85–97. Springer Berlin Heidelberg.

[25] Droubay, X. (1995). Palindromes in the Fibonacci word. *Information Processing Letters*, 55(4):217–221.

[26] Farach, M. (1997). Optimal suffix tree construction with large alphabets. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 137–143. IEEE.

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

[28] Fischer, J. (2011). Inducing the lcp-array. In *Workshop on Algorithms and Data Structures*, pages 374–385. Springer.

[29] Fujishige, Y., Tsujimaru, Y., Inenaga, S., Bannai, H., and Takeda, M. (2016). Computing DAWGs and Minimal Absent Words in Linear Time for Integer Alphabets. In Faliszewski, P., Muscholl, A., and Niedermeier, R., editors, *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, volume 58 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–14.

[30] Gawrychowski, P., Lewenstein, M., and Nicholson, P. K. (2014). Weighted ancestors in suffix trees. In *European Symposium on Algorithms*, pages 455–466. Springer.

[31] Gelfand, M. S. and Koonin, E. V. (1997). Avoidance of palindromic words in bacterial and archaeal genomes: a close connection with restriction enzymes. *Nucleic Acids Research*, 25(12):2430–2439.

[32] Gog, S., Beller, T., Moffat, A., and Petri, M. (2014). From theory to practice: Plug and play with succinct data structures. In *International Symposium on Experimental Algorithms*, pages 326–337. Springer.

[33] Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA.

[34] Harmston, N., Barešić, A., and Lenhard, B. (2013). The mystery of extreme non-coding conservation. *Philosophical Transactions of the Royal Society B*, 368(1632):20130021.

[35] Hile, S. E. and Eckert, K. A. (2004). Positive correlation between DNA polymerase α-primase pausing and mutagenesis within polypyrimidine/polypurine microsatellite sequences. *Journal of molecular biology*, 335(3):745–759.

[36] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65.

[37] I, T., Sugimoto, S., Inenaga, S., Bannai, H., and Takeda, M. (2014). Computing palindromic factorizations and palindromic covers on-line. In *Combinatorial Pattern Matching*, volume 8486 of *LNCS*, pages 150–161. Springer International Publishing.

[38] Iliopoulos, C. S., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., and Tsakalidis, A. (2006). The weighted suffix tree: an efficient data structure for handling molecular weighted sequences and its applications. *Fundamenta Informaticae*, 71(2, 3):259–277.

[39] Kociumaka, T., Pissis, S. P., and Radoszewski, J. (2016). Pattern Matching and Consensus Problems on Weighted Sequences and Profiles. In *International Symposium on Algorithms and Computation*, volume 64 of *LIPIcs*, pages 46:1–46:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

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

[41] Levinson, G. and Gutman, G. A. (1987). Slipped-strand mispairing: a major mechanism for DNA sequence evolution. *Molecular Biology and Evolution*, 4(3):203–221.

[42] Lodish, H., Berk, A., Darnell, J. E., Kaiser, C. A., Krieger, M., Scott, M. P., Bretscher, A., Ploegh, H., Matsudaira, P., et al. (2008). *Molecular cell biology*. Macmillan.

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

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

[45] Mantegna, R. N., Buldyrev, S. V., Goldberger, A. L., Havlin, S., Peng, C.-K., Simons, M., and Stanley, H. E. (1994). Linguistic features of noncoding DNA sequences. *Physical Review Letters*, 73(23):3169.

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

[47] Mignosi, F., Restivo, A., and Sciortino, M. (2002). Words and forbidden factors. *Theoretical Computer Science*, 273(1):99–117.

[48] Muhire, B. M., Golden, M., Murrell, B., Lefeuvre, P., Lett, J.-M., Gray, A., Poon, A. Y., Ngandu, N. K., Semegni, Y., Tanov, E. P., et al. (2014). Evidence of pervasive biologically functional secondary structures within the genomes of eukaryotic single-stranded DNA viruses. *Journal of Virology*, 88(4):1972–1989.

[49] Nong, G., Zhang, S., and Chan, W. H. (2009). Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09.*, pages 193–202. IEEE.

[50] Polychronopoulos, D., Krithara, A., Nikolaou, C., Paliouras, G., Almirantis, Y., and Giannakopoulos, G. (2014a). *Analysis and Classification of Constrained DNA Elements with n-gram Graphs and Genomic Signatures*, pages 220–234. Springer International Publishing, Cham.

[51] Polychronopoulos, D., Sellis, D., and Almirantis, Y. (2014b). Conserved noncoding elements follow power-law-like distributions in several genomes as a result of genome dynamics. *PloS one*, 9(5):e95437.

[52] Polychronopoulos, D., Weitschek, E., Dimitrieva, S., Bucher, P., Felici, G., and Almirantis, Y. (2014c). Classification of selectively constrained DNA elements using feature vectors and rule-based classifiers. *Genomics*, 104(2):79–86.

[53] Porto, A. H. and Barbosa, V. C. (2002). Finding approximate palindromes in strings. *Pattern Recognition*, 35(11):2581–2591.

[54] Rubinchik, M. and Shur, A. M. (2016). Eertree: An efficient data structure for processing palindromes in strings. In *International Workshop on Combinatorial Algorithms*, volume 9538 of *LNCS*, pages 321–333. Springer International Publishing.

[55] Rusinov, I., Ershova, A., Karyagina, A., Spirin, S., and Alexeevski, A. (2015). Lifespan of restriction-modification systems critically affects avoidance of their recognition sites in host genomes. *BMC genomics*, 16(1):1.

[56] Searls, D. B. (1992). The linguistics of DNA. *American Scientist*, 80(6):579–591.

[57] Svoboda, P. and Cara, A. D. (2006). Hairpin rna: a secondary structure of primary importance. *Cellular and Molecular Life Sciences CMLS*, 63(7-8):901–908.

Appendix A

Constructing the special-weighted strings

Example A.0.1. Given the weighted string X=

`a[(a,0.5),(b,0.5)][(b,0.5),(c,0.5)]ba[(a,0.5),(b,0.5)]c[(a,0.5),(c,0.5)]aa`

of length $n = 10$ and a cumulative weight threshold $1/z = 1/4$, we proceed as follows.

Figure A.1 shows the WI for X. The red path on the figure shows how we construct one of the special-weighted strings. We begin from a leaf node with index 0, read the path-label and have $\mathcal{Z}_0 = \texttt{aabba}$. Then we follow the suffix link of its parent and progress down to find the node with path-label `abba`, which is a leaf node with index 1. We cannot append any letters to $\mathcal{Z}_0$ from this node so we follow the suffix link of its parent and find the next node with path-label `bba`. Since this is not a leaf node, we continue by spelling letters downwards the tree to the leaf node with index 2. We append

the letters to $\mathcal{Z}_0$ and have $\mathcal{Z}_0 = \texttt{aabbaac}$. Then we follow the suffix link, find the next node with path-label baac, and spell downwards. Finally we have $\mathcal{Z}_0 = \texttt{aabbaacaaa}$.

Similarly, by following the yellow nodes we have $\mathcal{Z}_1 = \texttt{aacbaaccaa}$. By following the green nodes we have $\mathcal{Z}_2 = \texttt{abbbabcaaa}$ and by following the gray nodes we have $\mathcal{Z}_3 = \texttt{abcbabccaa}$. Therefore, the special-weighted strings of X: $\mathcal{Z}_X = \{\mathcal{Z}_0, \mathcal{Z}_1, \mathcal{Z}_2, \mathcal{Z}_3\} = \{\texttt{aabbaacaaa}, \texttt{aacbaaccaa}, \texttt{abbbabcaaa}, \texttt{abcbabccaa}\}$. Note that although we described the special-weighted strings construction one by one, we do not construct them separately. We begin from all the leaf nodes with index 0 and construct all special-weighted strings together. In some cases, we may reach the same node when extending two or more special-weighted strings. For example, if we follow both red and green nodes in Figure A.1, we reach the explicit node with path-label bba (colored by both red and green), which has two branches ac$ and bc$. In this case, we check the occurrence probability for each branch, and since both branches are z-valid, we associate these two branches to two special-weighted strings respectively. As a result, we append ac to the one (red) and bc to the other (green).

Fig. A.1 The WI for X and $1/z$ in Example A.0.1 (labels of edges to terminal nodes are appended with a letter for convenience).