# Advances in Stringology and Applications
# From Combinatorics via Genomic Analysis to Computational Linguistics

Alatabbi, Ali

*Awarding institution:*
King's College London

# Advances in Stringology and Applications:

## from Combinatorics via Genomic Analysis to Computational Linguistics.

Ali Alatabbi

DOCTORAL DISSERTATION

2014

**King's College London.**
**School of Natural & Mathematical Sciences.**
**Department of Informatics.**

# ADVANCES IN STRINGOLOGY AND APPLICATIONS:

## FROM COMBINATORICS VIA GENOMIC ANALYSIS TO COMPUTATIONAL LINGUISTICS.

**Ali Alatabbi**

**DOCTORAL DISSERTATION**

**November 2014**

Thesis Committee:

**Prof. Costas S. Iliopoulos**
*King's College London.*

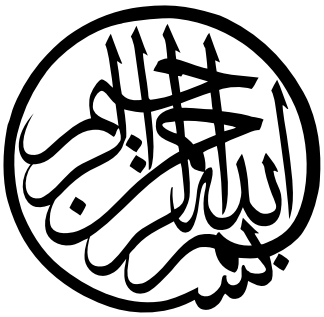
**Prof. Maxime Crochemore**
*King's College London.*

**Prof. Prudence Wong**
*University of Liverpool.*

**Prof. Marie-Pierre Béal**
*Université Paris-Est Marne-la-Vallée.*

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.


KING'S
College
LONDON



وَقُل رَّبِّ زِدْنِي عِلْمًا

# Abstract

Written text is considered as one of the oldest methods to represent knowledge. A text can be defined as a logical and consistent sequence of symbols which encodes information in a certain language. A straightforward example are natural languages, which are typically used by humans to communicate in spoken or written form.

Other underlying examples are DNA, RNA and proteins sequences; DNA and RNA are nucleic acids that carry the genetic instructions, specifies the sequence of the amino acids within proteins, regulate the development and functionality of living organisms specifies the sequence of the amino acids within proteins. Proteins are molecules consisting of one or more chains of amino acids participate in virtually every process within cells.

DNA and RNA can be represented as sequences of the nucleo-bases of their nucleotides and proteins and can be represented by the sequence of amino acids encoded in the corresponding gene. A natural problem which emerges when processing such sequences is determine weather a specific patterns occur within another string (known as exact string matching problem); as far as natural language texts are concerned, an important problem in computational linguistics is finding the occurrences of a given word or sentence in a volume of text; Similarly, in computational biology identifying given features in DNA sequences is a important of great significance, on the other side, one is often interested in quantifying the likelihood that two pairs of strings have the same underlying features based on explicit similarity/dissimilarity measurement (known as approximate string matching). Both instance of the string matching problem have been studied thoroughly since early 1960s.

This thesis contributes several efficient novel and derived solutions (algorithms and/or data structures), for complex problems which have been originated either out of theoretical considerations or practical problems, and study their experimental performance and compare the proposed solutions with some existing solutions.

Among the latter originated introduced solution several ones motivated by real-world problems in the fields of molecular biology and computational linguistics.

Despite the fact that studied problems and their proposed solutions differs in research motivation paradigm, yet still utilise similar tools and methodologies for solving the corresponding problems. For example the seminal “Aho-Corasick” Automaton is employed for finding a set of motifs in a biological sequence and detecting spelling mistakes in Arabic text. Similarly, employing the bit-masking trick to extend the DNA symbols to accelerate equivalency testing of degenerate characters in the same way to extend the Arabic alphabet to measure similarity between a stem and derived/inflected forms a given word.

**To:** ***Israa, Hamza, Nasrallah and Laila.***

# Acknowledgements

First of all, and more than anything else, I would like to acknowledge ***Prof. Costas S. Iliopoulos***, my first supervisor, for his support, excellent guidance, and inventive ideas throughout the time I was working on this thesis. I want also to acknowledge ***Prof. Maxime Crochemore***, my second supervisor, for his invaluable insights and advice.

***Prof. M. Sohel Rahman*** deserves a special thanks. The writing of this thesis would not have been possible, in the form in which it has happened, without his contribution.

Special acknowledgement is due to ***Prof. Jackie Daykin*** and ***Prof. William F. Smyth***, who have been a constant source of knowledge and inspiration, for their significant contribution to this thesis.

I wish to express my sincere appreciation to ***Prof. Prudence Wong*** and ***Prof. Marie-Pierre Béal*** for being my thesis examiners and for their valuable thoughts and suggestions.

My great gratitude to all of my co-authors, my colleges, the anonymous conference and journal referees and to those who have contributed to this thesis and supported me in one way or the other, who are not explicitly mentioned here, during this amazing journey.

Finally, I am sincerely thankful for my family for their love, understanding and unlimited support for making this experience possible for me.

I acknowledge the partial support of the INSPIRE Strategic Partnership Award, administered by the British Council, Bangladesh for the project titled "Advances in Algorithms for Next Generation Biological Sequences".

Ali Alatabbi

London, June 2015.

# Contents

# List of Figures

# Introduction

According to the Oxford Dictionary of Word Origins (2 ed.) the word '***algorithm***' originally meant the Arabic or decimal notation of numbers. It is a variant, influenced by Greek 'arithmos' - 'number'; of Middle English ***'algorism'***, which came via Old French from Mediaeval Latin; 'algorismus', derived from the Arabic name 'al-kwarizmi', which means "the man of Kwarizm" (now Khiva).

Abu Ja'far, Muhammad ibn Musa, Al-Kwarizmi (Baghdad, 780 – 850 AD) himself was a distinguished ninth century Muslim mathematician, astronomer, geographer and one of the greatest scientists of his faith. He was the author of widely translated works on algebra and arithmetic. He amalgamated Greek and Hindu knowledge and influenced mathematical thought to a greater extent than any other Mediaeval writer. He is considered the founder of analysis of algebra as distinct from geometry [Bre06].

His work on arithmetic was translated into Latin during the twelfth century and introduced the Hindu system of numeration to the Arabs and Europeans alike. His other work, the treatise Hisab al-jabr wal-muqabala (Algebra), which contains analytical solutions of linear and quadratic equations, is equally important.

The following, simple definition of ***'Algorithm'***, from The Dictionary of Computing, Oxford Reference (6th ed.) states that:

An '***Algorithm***' is a prescribed set of well-defined rules or instructions for the solution of a problem, such as the performance of a calculation, in a finite number of steps. Expressing an algorithm in a formal notation is one of the main parts of a program; much that is said about programs applies to algorithms, and vice versa. An effective algorithm is one that is effectively computable (see effective computability). The study of whether effective algorithms exist to compute particular quantities forms the basis of the theory of algorithms."

An equivalent intuitive notion of the modern algorithm is the question of whether a problem can be solved using an effective procedure.

Problems have to be formalised in order to be tackled systematically by computational algorithms. Therefore, all the problems have to be first formally defined where the given instance and question are stated clearly.

There are various ways to classify algorithms, each with its own merits.

- by implementation
- by design paradigm
- by field of study
- by complexity

Algorithms analysis is the area of computer science that studies the performance characteristics of a given algorithm and provides tools to analyze the correctness and efficiency of different methods of solutions.

The main objective for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications in order to compare it with other algorithms devised to solve the same problem. Moreover, the analysis of an algorithm helps understand it better, and can suggest informed improvements.

Algorithms should be analysed by employing independent mathematical techniques that analyze algorithms regardless of implementations, platforms, language, compiler or input instances. It is necessary to be content with algorithm validation. This process certifies, or verifies, that an algorithm will perform the calculation required of it. Once we have a correct algorithm for a problem, we have to determine the efficiency of that algorithm.

Algorithm analysis is concerned with comparing algorithms based upon the amount of computing resources that each algorithm uses. One want to be able to consider two algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer.

One branch of this study, average-case analysis, examines the average behaviour of the algorithm whereas worst-case analysis studies the behaviour when all circumstances are as unfavourable as possible (refereed as upper-bound). Meanwhile, the best-case analysis describe an algorithm's behavior under optimal conditions to calculate lower-bound on running time of an algorithm (the case that causes minimum number of operations to be executed).

As noted earlier, an algorithm is a generic, definite, precise step-by-step list of instructions for solving a problem. Therefore, in order to be able to consider two

algorithms and say that one is better than the other because it is more efficient in its use of those resources or perhaps because it simply uses fewer. We need a convention for analyzing and comparing algorithms based on the amount of time they require to execute, referred to as the "execution time" or "running time" of the algorithm, and based on the space requirements, the amount of space in memory an algorithm requires to solve the problem.

From this perspective, efficiency measures depend on the following:

Firstly, what we define to be a 'step' or "unit of work", which we define here as the number of elementary operations taken by an algorithm, or its running time, and "unit of space" is essentially the number of memory cells which an algorithm needs.

If each of these steps is considered to be a basic unit of computation, then the running time for an algorithm can be expressed as the number of steps required to solve the problem. It is important to quantify the number of operations or steps that the algorithm will require. For the analysis to measure up the actual execution time, the time required to perform a 'step' must be guaranteed to be bounded above by a constant.

Often, there is a time-space-tradeoff involved in solving a problem, that is, it cannot be solved as fast as possible using as small as possible memory consumption. A compromise has to be made to compensate computing time for memory consumption or vice versa, can differ significantly depending on the algorithm chosen and how it was configured.

Secondly, acquiring a prospective for the relative growth of functions to understand their behavior. The growth rate of a function describes the rate at which the value of the function changes as the size of its input increases. Order of growth is always expressed in terms of the size of the problem without stating what the problem is. The growth rate gives the general shape of the function, rather than its specific value.

Rather than specifying the exact relation between an algorithm's input and its running time, describing the growth rate, of the running time, for large input sizes provides enough evidence to distinguish between a good and a bad algorithm.

An optimal algorithm is an algorithm for which both the upper-bound of the algorithm and the lower-bound of the problem are asymptotically equivalent.

Meaning that one cannot hope for an algorithm that runs in time that is asymptotically less than the lower bound of the problem in the worst case. More formally,

suppose that we have a lower-bound theorem showing that a problem requires $\Omega(f(n))$, with respect to a particular resource, to solve for an instance of that problem of size $n$. Then, an algorithm which solves the problem in $O(f(n))$ time/space is said to be asymptotically optimal.

In practice, other considerations beside asymptotic analysis are important when choosing between algorithms. Although asymptotically optimal algorithms are important theoretical results, such algorithm might not be favored one in a number of practical situations for many reasons, such as, complexity, hardware/software limitations, there could be sub-optimal or heuristic algorithms exist that make better use of certain aspects of the considered problem, and outperform an optimal algorithm [Gus97].

Data structures provide ways of storing and organising data, and can inhabit both in the main and in secondary memory, in order to be used efficiently. Different kinds of data structures are suited to different kinds of applications and some are highly specialised to specific tasks or specific types of data. Many algorithms apply directly to a specific data structures. Sometimes, efficient data structures are a key to designing efficient algorithms and choice of an appropriate data structure can influence the design of an algorithm significantly.

Two famous index data structures used for large strings are the inverted k-mer index and the family of indexes related to suffix trees (including, most importantly, suffix arrays [MM90, PST07] and Burrows-Wheeler index [BW94]). Inverted indexes are very fast in practice but are less suited for approximate matching, although they have been used for this purpose. Some data structures such as CSA (Compressed Suffix Array) [Lip05] and FM–Index [FM05] present an improvement over classical data structures taking into account the entropy of a sequence to produce a compressed index of the sequence using techniques such as block addressing. Another example is a DAWG (directed acyclic word graph), which is used to determine quickly whether or not a particular word is in a set of words and should be considered any time a search through a large lexicon is needed [AJ88].

One of the properties that are accepted as requirements for an algorithm is to halt on every input, which implies that each instruction requires a finite amount of time, and the input has a finite length [Sto72],[Knu10]. It is also required that the output has to be unique for each input, that is, the algorithm is deterministic in the sense that when the algorithm is re-initiated with a particular input the same set of instructions should

be executed, otherwise the algorithm is called non-deterministic or randomized.

There can be degrees of deterministic behavior. For instance, an algorithm that uses random numbers is not usually considered deterministic. However if the random numbers come from a pseudo-random number generator, the behavior may be deterministic.

Different algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be far more significant than differences that arise from hardware and software advancement.

The family of ***"string algorithms"*** (or ***"stringology"***, the name was coined in 1984 by computer scientist Zvi Galil [Str13], refereed to algorithms and data structures used for string processing) provides a generic solution of string-related algorithms without stating what the problem is.

The main purpose of "string algorithms" is to manipulate strings of symbols, to compare them, count them, examine attributes and properties, and to perform a variety of format transformations in an effective and efficient way. A vital role played by Strings Algorithms is that they exploit (hidden) characteristics of any given texts.

Through these topological hypotheses, string algorithms significantly speed up processing cost (time/space). String algorithms are often of low complexity, but are intricate and difficult to prove [Gus97]. Many algorithms archive an optimal linear-time complexity corresponding to a single pass scanning of the input string. However, proof of correctness for such algorithms is usually complex.

A text, whether written or spoken, is, of course, one of the oldest methods used to represent and preserve knowledge. Text can be defined as a logical and consistent sequence of symbols encoding information in a certain language.

Natural languages provide straightforward examples of texts used by humans to communicate in spoken or written forms. In natural languages, text comprises a number of words. The term "word" may refer to a spoken word or to a written word, or sometimes to the abstract concept behind either of these. Spoken words are made up of units of sound called phonemes, and written words are made up of symbols called graphemes, such as the letters of the English alphabet.

In linguistics, a word is the smallest element that may be uttered in isolation with semantic or pragmatic content (with literal or practical meaning). Traditionally, words are the smallest of the units that make up a sentence, and marked as such in writing.

Words combine to make phrases In practice, words are established by various criteria. They are generally the smallest units that can form an utterance on their own.

Musicology, the systematised study of music and musical style, particularly in the realm of historical research, is another example whereby written text - a score - and sounds, i.e., music, represents another type of text.

Other underlying examples are DNA, RNA and protein sequences.

DNA and RNA are nucleic acids that carry the genetic instructions, specify the sequence of the amino acids within proteins and regulate the development and functionality of living organisms. Proteins are molecules consisting of one or more chains of amino acids which participate in virtually every process within cells.

DNA and RNA can be represented as sequences of the nucleo-bases of their nucleotides and proteins and can be represented by the sequence of amino acids encoded in the corresponding gene.

In computer science, a string is generally understood as a data type and is often implemented as an array of bytes (or words) that stores a sequence of elements (treated as a single entity), typically characters, using some character encoding [Wik14].

Despite the fact that the examples presented above seem very divergent, there is an operation that arises in most applications handling these types of sequences. This operation is called "text search" and consists of finding all positions on the text where a given pattern appears. This operation serves as a basis for building more complex and meaningful operations used in implementation of practical software running under various operating systems. Furthermore they represent programming methodologies that serve as paradigms in other fields of computer science. Finally, they also play an important role in theoretical computer science by providing challenging problems.

A natural problem which emerges when processing such sequences lies in determining whether specific patterns occur within another string. This problem is known as the "exact string matching problem" [CL04, FL10].

As far as natural language texts are concerned, an important problem in computational linguistics is finding the occurrences of a given word or sentence in a volume of text. Similarly, in computational biology identification of conserved features in a set of DNA or protein sequences represents a problem that is of great significance.

The problem in its most general form is to find the positions in a text where a given pattern occurs allowing a limited number of errors in the matches. This demands that

we have a notion of nearness or proximity between a pair of strings. This problem, which is known as "approximate string matching", is no less important. In this case we have to provide a definition for the distance function which measures the similarity between two strings [Nav01].

Various types of similarity measurement have been defined and studied in the literature, each application/domain uses a different error model which defines how different two strings are and set the criteria to choose the "nearest" one, for example, errors (edit distance [Lev66, Dam64, FW74, Ukk85], Hamming distance [Ham50], Longest Common Subsequence [NW70, San72, Sel74, Hir77]), $q$-gram distance [Ukk92], wild cards (or don't cares) [FP74, Pin85, MBY91], rotations [FNU02, FU98], scaling [ALV90] and permutations [Knu68].

"Approximate string matching" is based on the remarkable paper by Wagner and Fischer [FW74]. The "Exact string matching" is based on the two historical papers by Knuth, Morris and Pratt [KMP77] and by Boyer and Moore [BM77]. While the "Multiple string matching" was by Aho and Corasick [AC75].

The ideas behind these similarity measurements between strings is to make it small when one of the strings is likely to be an erroneous variant of the other under the error model in use.

The approximate occurrence can be defined using a variety of distance metrics over strings. Some popular metrics are Hamming distance, unit-cost edit distance and general edit distance based on a substitution cost matrix.

Depending on which string is given, the pattern or the text is available first. The solution can be based algorithms, automaton or combinatorial properties of strings which are usually implemented to process the pattern and solve the problem. The notion of indexes represented by data structures, such as tree or automaton, is used in the second types of solutions.

Both instances (exact and approximate) of the string matching problem have been studied extensively since the early 1960s (see [BY89],[CR94],[Smy13],[Nav01] and the references there).

The study of combinatorics on words started at the beginning of the 20th century with the work of the Norwegian Mathematician Axel Thue [Thu06]. His paper considered the founding document of combinatorics on words, thousands of research papers have been written by mathematicians and (over the last half-century) also computer

scientists, many algorithms that have been proposed that relate in some way to string regularities , or its variants, in strings since then. However, combinatorics of repetitions remains intricate area, full of open problems.

There are several types of regularities in strings: “repetition“ (is a repeating substrings (factors) that are constrained to be adjacent or otherwise those that may be nonadjacent or overlapping (“repeat”), square, period, run (or maximal periodicity), cover, seed, palindrome , etc (see [Smy13]).

Studying regularities in strings is important both from theoretical and practical point of view, these are studied in Chapter II - Structured regularities on strings.

Repetitions and periods in strings constitute one of the most fundamental areas of string combinatorics, detecting such regularities is an important element of several fields such as pattern matching, data compression, formal language theory and automata theory, to mention a few (see [Lot05, Gus97, CHL07] and the references there). Particularly, string regularities play an important role of applications in musicology, computational linguistics and computational biology.

Pattern matching algorithms have to cleverly exploit such regularities in order to efficient. In Particular, we mention regularities that characterize the string as well as (usually) all of its prefixes. These regularities are extensions of the idea of a “failure function” [AC75], or “border array” [Smy03]) that permits all the periods of every prefix of a given string to be compactly expressed by a single array of integers [Smy13].

Similarly, Analysing rotations of strings can be useful for algorithms whose operation depends on rotations of strings and their lexicographic ordering. One such algorithm is the block-sorting transformation known as Burrows-Wheeler transform (BWT) [BW94] used to bring repeated characters together as a preliminary to compression.

In recent times we have witnessed a rapid growth in the volume of digital information. This growth may be ascribed to the overwhelming increase in research, development, and investment in ICT - Information and Communication Technology - combined with the emerging interdisciplinary field of Bioinformatics and its allied disciplines.

The work presented in this thesis looks at data growth and new challenges, in existed and/or newly- discovered fields, that emerges as a result of such growth.

The exponential growth in the amount of digital data has resulted in the creation of

text volume on a scale that is unprecedented. The amount of electronic data available now is immense and continues to grow rapidly, in part due to the phenomenal growth of the Internet, but also as a result of increases in other data sources such as high throughput sequencing machines.

Most of this information exists in the form of text, that is, sequences of symbols representing natural language, music, source code, time series, biological sequences and many others.

Following on from the acquisition of data, attention has shifted to focus on analysing and making use of the collected data.

Rapid development in DNA sequencing technologies has caused a dramatic growth in the size of publicly available sequence databases with such data.

DNA sequencing has become incredibly fast and cost-effective to such an extent that sequencing individual genomes will soon become a common task, making querying and storing such sets of data especially important task.

The genomic revolution has changed how molecular biologists discover the structure, function and role of genes and protein sequences. Understanding relationships between unknown gene/protein sequence and known sequences is a key to assigning its function.

This presents interesting research challenges regarding the efficient storage and ability to access the data due to the highly repetitive nature of the sequences.

The human genome has about 3 billion DNA base pairs (bps), consisting of 23 chromosomes with lengths ranging from about 33 to 247 million bps.

DNA sequences within the same species are highly repetitive and often will only have a few differences. In large data sets, for example, such as genomes from a single species, it is noted that each entry only differs from another by a very small number of variations [CFMPN10]. This leads to a large set of data with a great deal of redundancy and repetition.

A repetitive sequence collection is one where a base sequence is repeated many times with small variations. Examples of such collections appear in large sets of sequence reads (usually obtained from Solid, Illumina or 454 sequencers) or complete genome sequences of human individuals where the differences can be expressed by a list of basic edit operations, the rest is common to all humans..

Algorithms that operate on molecular sequence data (strings) are at the heart of

computational molecular biology wide spectrum of string techniques.

Algorithms and Data structures, such as the ones introduced in Chapter III - Improved solutions for molecular biology of this thesis, address some of these challenges.

Many challenges in Arabic computational linguistics exist, these are studied in Chapter IV - Challenges in Arabic computational linguistics. The influence of the Arabic language, which belongs to the Semitic language family originated in the Arabian Peninsula in pre-Islamic times, and which spread rapidly across the Middle East, has been of great significance. Arabic has been an important source of vocabulary for many other languages and is one of the official languages of the United Nations, is the official language in over 29 countries and is the sixth most used language in the world, spoken by over 360 million people. Arabic has a very rich and complex morphology. Its appropriate Morphological processing is very important for Information Retrieval, Text Processing, Machine Translation and Spell Checking processes.

Yet despite its importance globally, efforts to improve Arabic information search and retrieval, compared with other languages, are limited and, at best, modest. One of the main barriers to text processing advancements in Arabic is the language"s complicated morphological structure. Arabic also has complicated syntactic properties which make it a difficult language to master for non-native speakers.

Among these properties the complex structure of the Arabic word, the agglutinative nature, lack of vocalisation, the segmentation of the text, the linguistic richness... all these factors unite to create barriers.

In the context of linguistics, morphology is the study of word forms. In formal language theory, the symbols for representing words are an inseparable part of the definition of the language. In human languages, the concept is a little different a statement can have multiple representations, depending on the means of communication and the conventions for recording it. Arabic morphology is well-known for its rich and complex nature.

Morphological complexity negatively affects performance of spell checking systems. Arabic morphology has a multi-tiered structure and applies non-concatenative morphotactics. Words in Arabic are originally formed through the amalgamation of roots and patterns.

The objective of this thesis is that it aims to develop efficient algorithms on strings

in order to deal with pure theoretical problems or other problems encountered in such fields as Bioinformatics and Computational linguistics.

This can be achieved by studying the problems' statistical behaviour which will lead to a better understanding of and eventual solution to the problems.

Despite the fact that the problems investigated, as well as their proposed solutions, differ in research motivation paradigm, analogous mechanisms and structures are employed for solving these problems.

For example, the seminal string matching algorithm "Aho-Corasick" Automaton is employed for finding a set of motifs in a biological sequence as well as in detecting spelling mistakes in Arabic text.

Similarly, the bit-masking trick is used in order to extend both the DNA symbols and Arabic alphabet: to accelerate equivalency testing of degenerate sequences in the DNA and to measure similarity between stem and surface forms of a given word in Arabic.

Another example, the surprisingly wide range of problems that can be tackled by utilising Lyndon words: Lyndon words proved to be useful for constructing bases in free Lie algebras [Reu93], constructing de Bruijn sequences [FM78], computing the lexicographically smallest or largest substring in a string [AC95], succinct suffix-prefix matching of highly periodic strings [NS13] and string matching [CP91, BGM11]. With wider ranging applications include the Burrows-Wheeler transform and data compression [GS12], musicology [Che04], bioinformatics [DR04], and in relation to cryptanalysis [Per05].

This thesis contributes several efficient novel and derived solutions (algorithms and/or data structures), for complex problems which arise either out of pure theoretical considerations (Chapter II - Structured regularities on strings) or practical application in molecular biology (Chapter III - Improved solutions for molecular biology) and computational linguistics in Arabic (Chapter IV - Challenges in Arabic computational linguistics). This thesis also studies the experimental performance of these problems and compares the proposed solutions with those that already exist.

# Organization

This dissertation is divided into five chapters as follows.

**Chapter I - Notions and Definitions:**

Provides the readers with the basic notions required to properly follow the work and results presented in the subsequent chapters; in particular, it introduces the basic definitions and notations on alphabet and strings, string similarity metrics, asymptotic notation and also describe some elementary data structures.

**Chapter II - Structured regularities on strings:**

Focuses on regularities that characterize the string as well as, usually, all its prefixes. some of these regularities are extensions of the idea of a "failure function", in particular, string periodicity properties. Furthermore, we study repetitive structures and inherent patterns such as Lyndon words, Abelian strings and palindromes.

- We start by describing ***indeterminate*** strings, in the most general sense, as the basic combinatorial object and introduce the first algorithm to reverse engineer a data structure (prefix array) to a string in its full generality. We show how to construct a lexicographically least indeterminate string on a minimum alphabet from its analogous prefix array.

- Next, we describe a simple linear-time/space algorithm to compute the cover array of regular string directly from its prefix table. Then we describe extensions of these algorithms to indeterminate strings.

- We consider the problem of finding the length of the ***Longest Common Abelian Factor (***`LCAF`***)*** between two strings and present a quadratic running time algorithm for the `LCAF` problem and a sub-quadratic running time solution for the binary string case, both having linear space requirement. Furthermore, we present a variant of the quadratic solution that is experimentally shown to achieve a better time complexity of $\mathcal{O}(\sigma\ n\ \log n)$, where $\sigma$ is the alphabet of cardinality, i.e., $\mathcal{O}(n\ \log n)$ for a constant alphabet.

- We propose a new linear-time algorithm that computes the ***Maximal Palindromic Factorization (MPF)*** of a string, that is a factorization of a string where the factoriza-

tion set is the set of all center-distinct maximal palindromes of a string (the algorithms is evaluated with respect to the length of the given string).

- We study Lyndon structures related to the Burrows-Wheeler Transform (BWT). We compute the quadratic factorization of all rotations of an input string and the BWT of a Lyndon substring. From the factored rotations we introduce the ***Lyndon fountain***.

- We combine the well-known concepts of Lyndon words, borders and suffix arrays to introduce the ***Lyndon Border Array (LBA)*** and the ***Lyndon Suffix Array (LSA)***. We present linear time algorithms for computing these two interesting data structures.

- In the final article of this chapter we focus on a total (but non-lexicographic) ordering of strings called $V$-order. We devise a new linear-time algorithm for computing the $V$-comparison of two finite strings.

**Chapter III - Improved solutions for molecular biology:**

In this chapter we design and analyse algorithms/data structures on strings with applications in molecular biology.

- We present ***SimpLiSMS***, a simple, lightweight and fast algorithm for searching structured motifs. We introduce the concept of a ***search context***, and makes use of a simple data structure (***Map***) to identify the valid positions of each character with respect to its preceding character according to the distance constraints of the structured motif. Our experiments show excellent performance of ***SimpLiSMS***. Furthermore, We introduce a parallel version of ***SimpLiSMS*** which runs even faster.

- The next article describes another solution to the compression of a set of genomic sequence data-set problem, we propose an indexing structure for highly repetitive collections of genomic data based on a multilevel $q$-gram model, utilising a differential compression method that is based on the locations and types of differences between each sequence in the collection and its reference sequence. In particular, the proposed algorithm accommodates variations that may occur in the target sequence with respect to the reference sequence.

**Chapter IV - Challenges in Arabic computational linguistics:**

This chapter is dedicated for exploring complex problems in the domain of Natural Language Processing for Arabic language to present hybrid solutions by combining

efficient computational methods and language morphological rules to build Language-Ware software for Arabic language.

We have defined two main objectives for this study, to formalize an elegant linguistic description of Arabic words and to develop a set of computational resources making use of this linguistic formalization.

- The first article in this chapter describes the construction of a lexicon and a morphological description for standard Arabic. We present a large-scale system that performs morphological analysis and generation of Arabic words.

- We propose a new stemming technique and produce software implementation (Arabic Morphological Analyzer called ***"AMA"***), for the proposed technique that tries to determine the root and/or the stem of a word representing the semantic core of this word according to Arabic language morphology analysis and Arabic language syntax.

- We build an expert system for Arabic spelling correction using a generative noisy channel model, that goes beyond the primitive edit distance to compute (learnable edit distance) - by presenting new conditioning factors - the optimal costs of a set of string edit rules based on the morphological characteristics and letters adjacency probabilities of Arabic words.

The algorithm learns the model parameters using a training data-set consisting of pairs of erroneous and their correct words to build an error model. The algorithm uses dynamic programming to calculate the characters edit distance/rules to learn the minimum total cost of transforming one string into another. Also we present the notion of single candidate errors and introduce a novel method for detecting and correcting many such errors that cannot be detected by existing techniques.

- Finally, we introduce a new model, Degenerate Finite-State Automaton (`DeFSA`), aimed at facilitating the expression of various non-concatenative morphological phenomena in an effective and efficient way. We show how to utilise the new data structure and its DAWG implementation to build an Arabic lexicon.

**Chapter IV - Concluding Remarks:** Summarises the results of the work presented in the previous chapters, conclusions are drawn and finally a set of related open problems are presented.

# List of Publications

[1] **Ali Alatabbi**, Shuhana Azmin, Md. Kawser Habib, Costas S. Iliopoulos and M. Sohel Rahman, **SimpLiSMS: A Simple, Lightweight and Fast Approach for Structured Motifs Searching**, In Francisco M. Ortuño Guzman and Ignacio Rojas, editors, Bioinformatics and Biomedical Engineering – Third International Conference – IWBBIO 2015. Proceedings, Part II, volume 9044 of Lecture Notes in Computer Science, pages 219 – 230. Springer, 2015.

[2] **Ali Alatabbi**, M. Sohel Rahman, and William F. Smyth, **Computing Covers Using Prefix Tables**, In *Discrete Applied Mathematics*, DOI: 10.1016/j.dam.2015.05.019, to appear (2015).

[3] **Ali Alatabbi**, Costas S. Iliopoulos, Alessio Langiu, and M. Sohel Rahman, **Algorithms for Longest Common Abelian Factors**, In *International Journal of foundations of Computer Science*, to appear (2015).

[4] **Ali Alatabbi**, Jacqueline W. Daykin, M. Sohel Rahman and William F. Smyth, **Simple linear comparison of strings in V–order**, In *Fundamenta Informaticae*, DOI: 10.3233/FI-2015-1228, vol. 139(2), pages 115 – 126, 2015.

[5] **Ali Alatabbi**, M. Sohel Rahman, and William F. Smyth, **Inferring an indeterminate string from a prefix graph**, In *Journal of Discrete Algorithms*, DOI: 10.1016/j.jda.2014.12.006, issue 2, issn 1570 – 8667, volume 32, number 0, pages 6 – 13, StringMasters 2012 & 2013 Special Issue (Volume 2), 2015.

[6] **Ali Alatabbi**, Jacqueline W. Daykin, M. Sohel Rahman, and William F. Smyth, **Simple linear comparison of strings in V–order (extended abstract)**, In Pro-

ceedings of *8th International Workshop on Algorithms & Computation – WALCOM 2014*, LNCS 8344, pages 80 – 89, 2014.

[7] **Ali Alatabbi**, Costas S. Iliopoulos, and M. Sohel Rahman, **Maximal Palindromic Factorization**, In Proceedings of *The Prague Stringology Conference – PSC 2013*, pages 70 – 77, 2013.

[8] **Ali Alatabbi**, Costas S. Iliopoulos, Alessio Langiu, and M. Sohel Rahman, **Computing the Longest Common Abelian Factor**, In Proceedings of *The 14th Italian Conference on Theoretical Computer Science – ICTCS 2013*, pages 215 – 221, 2013.

[9] **Ali Alatabbi**, and Costas S. Iliopoulos, **Morphological analysis and generation of Arabic language**, In *International Journal of Information Technology & Computer Science, IJITCS*, issue 2, ISSN 2091 – 0266, volume 3, pages 20 – 28, 2012.

[10] **Ali Alatabbi**, Carl Barton, and Costas S. Iliopoulos, **On the repetitive collection indexing problem**, In Proceedings of *IEEE International Conference on Bioinformatics and Biomedicine Workshops – BIBM 2012*, pages 682 – 687. IEEE, 2012.

[11] **Ali Alatabbi**, and Costas S. Iliopoulos, **Novel Arabic Language Stemmer**, In Proceedings of *The International Conference on Computer Science, Engineering & Technology – ICCSET'12*, pages 20 – 28, 2012.

[12] **Ali Alatabbi**, Carl Barton, Costas S. Iliopoulos, and Laurent Mouchard, **Querying highly similar structured sequences via binary encoding and word level operations**, In Proceedings of *Artificial Intelligence Applications and Innovations – AIAI 2012 International Workshops*, pages 584 – 592, 2012.

[13] **Ali Alatabbi**, Maxime Crochemore, Jacqueline W. Daykin, and Laurent Mouchard, **Lyndon fountains and the Burrows–Wheeler transform**, In Proceedings of *International IT Conference & Exhibition – CUBE'12,*, pages 441 – 446, 2012.

# Chapter I

# Notions and Definitions

This chapter establishes the foundation needed to read most of the following chapters. It also provides an overview of the field and touches on some aspects that later chapters will discuss in more depth. Some mathematical concepts will be used throughout the chapters of this thesis. Therefore, this introductory chapter defines these concepts. Section 1.1 introduces basic strings and alphabets characteristics. String metrics are presented in 1.2 and some fundamental data structures are discussed in section 1.3. Finally, section 1.4 briefly reviews some notions from complexity theory and surveys the most important complexity classes. These notions will be frequently used in later discussions.

## 1.1 Alphabets and Strings

An ***alphabet*** $\Sigma$ is a finite non-empty set whose elements are called ***symbols*** (or ***characters***). The cardinality of an alphabet, denoted by $|\Sigma|$ expresses the number of distinct ***characters*** in the ***alphabet***.

The set of all the strings on the ***alphabet*** $\Sigma$ is denoted by $\Sigma^*$. The set of all non-empty ***strings*** over the ***alphabet*** $\Sigma$ is denoted by $\Sigma^+$. The ***empty string*** is the ***empty sequence*** (i.e., of zero length) and is denoted by $\epsilon$; we write $\Sigma^* = \Sigma^+ \cup \epsilon$.

A ***string*** or ***sequence*** is a succession of zero or more symbols drawn from an ***alphabet*** $\Sigma$. A ***string*** $\boldsymbol{s}$ of length $|\boldsymbol{s}| = n$ is represented by $\boldsymbol{s}[1\,..\,n]$, where $\boldsymbol{s}[i] \in \Sigma$ for $1 \leq i \leq n$. The $i$-th symbol of a ***string*** $s$ is denoted by $\boldsymbol{s}[i]$. We denote by $\boldsymbol{s}[i\,..\,j]$ the substring of $\boldsymbol{s}$ that starts at position $i$ and ends at position $j$.

For clarity, we may represent a ***string*** by $s_1 \cdot\cdot\, s_n$, also denote the $i$-th symbol of a ***string*** $\boldsymbol{s}$ by $s_i$, and similarly denote by $s_i \cdot\cdot\, s_j$ the substring of $\boldsymbol{s}$ that starts at position $i$ and ends at position $j$.

Also, some times we denote by $s[i]$, for all $0 \leq i < |\boldsymbol{s}|$, the symbol at index $i$ of $\boldsymbol{s}$. Each index $i$, for all $0 \leq i < |\boldsymbol{s}|$, is a position in $\boldsymbol{s}$ when $\boldsymbol{s} \neq \varepsilon$. It follows that the $i$-th symbol of $\boldsymbol{s}$ is the symbol at position $i-1$ in $\boldsymbol{s}$, and that $\boldsymbol{s} = \boldsymbol{s}[0\,..\,|\boldsymbol{s}|-1]$.

A ***string*** $\boldsymbol{w}$ is a substring (or *factor*) of a ***string*** $\boldsymbol{s}$ if there exist two strings $\boldsymbol{u}$ and $\boldsymbol{v}$, such that $\boldsymbol{s} = \boldsymbol{uwv}$, where $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$. Conversely, $\boldsymbol{s}$ is called a super-string of $\boldsymbol{w}$.

For a substring $\boldsymbol{w}$ of $\boldsymbol{s}$, $\boldsymbol{uwv}$ for $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$ is an extension of $\boldsymbol{w}$ in $\boldsymbol{s}$ if $\boldsymbol{uwv}$ is a substring of $\boldsymbol{s}$; $\boldsymbol{wv}$ for $\boldsymbol{v} \in \Sigma^*$ is the right extension of $\boldsymbol{w}$ in $\boldsymbol{s}$ if $\boldsymbol{wv}$ is a substring of $\boldsymbol{s}$; $\boldsymbol{uw}$ for $\boldsymbol{u} \in \Sigma^*$ is a left extension of $\boldsymbol{w}$ in $\boldsymbol{s}$ if $\boldsymbol{uw}$is a substring of $\boldsymbol{s}$.

We call a ***string*** $\boldsymbol{y}$ a subsequence of $\boldsymbol{x}$ (or $\boldsymbol{x}$ is a super-sequence of $\boldsymbol{y}$) if $\boldsymbol{y}$ is obtained by deleting zero or more symbols at any positions from $\boldsymbol{x}$. For example $ace$ is a subsequence of $aabcdef$.

For a given set $\mathcal{S}$ of strings, a ***string*** $\boldsymbol{x}$ is called a common super-sequence of $\mathcal{S}$ if $\boldsymbol{x}$ is a super-sequence of every ***string*** $\boldsymbol{s} \in S$.

The ***string*** $\boldsymbol{xy}$ is a concatenation of two strings $\boldsymbol{x}$ and $\boldsymbol{y}$. The concatenations of $k$ copies of $\boldsymbol{x}$ is denoted by $\boldsymbol{x}^k$.

For two strings $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ and $\boldsymbol{y} = \boldsymbol{y}[1 \cdot\cdot\, m]$, such that $\boldsymbol{x}[n-i+1\,..\,n] = \boldsymbol{y}[1\,..\,i]$ for some $i \geq 1$, the ***string*** $\boldsymbol{z} = \boldsymbol{x}[1\,..\,n]\boldsymbol{y}[i+1\,..\,m]$ is a superposition of $\boldsymbol{x}$ and $y$ with $i$ overlap.

The number of occurrences of the letter $\sigma_i$ in ***string*** $\boldsymbol{s}$ is denoted $|\boldsymbol{s}|_{\sigma_i}$.

The conjugacy class $[\boldsymbol{s}]$ of ***string*** $\boldsymbol{s} \in \Sigma^n$ is the set of all words $s[i]s[i+1]\cdots s[n-1]s[0]\cdots s[i-1]$, for $0 \leq i \leq n-1$.

If $\boldsymbol{s}$ is not the power of a shorter word, then $\boldsymbol{s}$ is said to be primitive and has exactly $n$ conjugates.

**Definition 1 (Cyclic shift)** *A* ***string*** $\boldsymbol{y} = \boldsymbol{y}[0\,..\,n]$ *is a cyclic rotation of* $\boldsymbol{x} = \boldsymbol{x}[0\,..\,n]$ *if* $\boldsymbol{y}[0\,..\,n] = \boldsymbol{x}[i\,..\,n]\boldsymbol{x}[0\,..\,i-1]$ *for some* $1 \leq i \leq n$ *(for* $i = 1, \boldsymbol{y} = \boldsymbol{x}$*).*

*Equivalently, we say two words* $\boldsymbol{x}$*,* $\boldsymbol{y}$ *are* conjugates *if there exist words* $\boldsymbol{u}$*,* $\boldsymbol{v}$ *such that* $\boldsymbol{x} = \boldsymbol{uv}$ *and* $\boldsymbol{y} = \boldsymbol{vu}$*.*

*Conjugacy is thus an equivalence relation. The conjugacy class of a word of length* $n$ *and period* $p$ *has* $p$ *elements if* $p$ *divides* $n$ *and has* $n$ *elements otherwise.*

**Definition 2 (Lexicographical order)** *Consider the finite totaly ordered alphabet* $\Sigma$*. The lexicographical order is defined as follows. Given two strings* $\boldsymbol{x}$*,* $\boldsymbol{y}$*, we have* $\boldsymbol{x} < \boldsymbol{y}$ *if* $\boldsymbol{x}$ *is a proper prefix of* $\boldsymbol{y}$ *or if there exist factorization* $\boldsymbol{x} = \boldsymbol{u}a\boldsymbol{x}'$ *and* $\boldsymbol{y} = \boldsymbol{u}b\boldsymbol{y}'$ *with* $a, b \in \Sigma$ *and* $a < b$*. Note that* $\boldsymbol{x} < \boldsymbol{y}$ *in the radix order if* $|\boldsymbol{x}| < |\boldsymbol{y}|$ *or* $|\boldsymbol{x}| = |\boldsymbol{y}|$ *and* $\boldsymbol{x} < \boldsymbol{y}$ *in the lexicographical order.*

A substring $\boldsymbol{w}$ of $\boldsymbol{s}$ is called repetition in $\boldsymbol{s}$, if $\boldsymbol{s} = \boldsymbol{uw}^k\boldsymbol{v}$, where $\boldsymbol{u}, \boldsymbol{w}, \boldsymbol{v}$ are substrings of $\boldsymbol{s}$ and $\boldsymbol{k} \geq 2, |\boldsymbol{w}| \neq 0$.

For example if $\boldsymbol{x} = aababab$, then $a$ (appearing in positions 1 and 2) and $ab$ (appearing in positions 2,4 and 6) are repetitions in $\boldsymbol{w}$; in particular $a^2 = aa$ is called a $square$ and $(ab)^3 = ababab$ is called a cube.

A ***string*** $\boldsymbol{u}$ is a prefix of $\boldsymbol{w}$ if $\boldsymbol{w} = \boldsymbol{uv}$ for $\boldsymbol{v} \in \Sigma^*$, Similarly, $\boldsymbol{v}$ is a suffix of $\boldsymbol{w}$ if $\boldsymbol{w} = \boldsymbol{uv}$ for $\boldsymbol{u} \in \Sigma^*$.

The prefix $\boldsymbol{u}$ (respectively suffix $\boldsymbol{v}$ ) is a proper prefix (suffix) of a word $\boldsymbol{w} = \boldsymbol{uv}$ if $\boldsymbol{w} \neq \boldsymbol{u}, \boldsymbol{v}$.

### 1.1.1 Borders and Periods

A substring $\boldsymbol{u}$ is called a period of a ***string*** $\boldsymbol{s}$ (notation: $\boldsymbol{u} = per(\boldsymbol{s})$), if $\boldsymbol{s}$ can be written as $\boldsymbol{s} = \boldsymbol{u}^k\boldsymbol{u}'$ where $k \geq 1$ and $\boldsymbol{u}'$ is a prefix of $\boldsymbol{u}$. The shortest period of $\boldsymbol{s}$ is called the period of $\boldsymbol{s}$. For example if $\boldsymbol{s} = abcabcab$, then $abc, abcabc$ and the ***string*** $\boldsymbol{s}$ itself are periods of $\boldsymbol{s}$ while $abc$ is the period of $\boldsymbol{s}$.

In other words, a ***string*** $\boldsymbol{u}$ is a period of $\boldsymbol{s}$ if $\boldsymbol{s}$ is a prefix of $\boldsymbol{u}^k$ for some $k$,or equivalently if $\boldsymbol{s}$ is a prefix of $\boldsymbol{us}$.

Moreover a ***string*** is said to be primitive if it cannot be be written as $\boldsymbol{u}^k$ with $\boldsymbol{u} \in \Sigma^+$ and $k \geq 2$.

A set of strings that are both prefixes and suffixes of $\boldsymbol{s}$ are called borders of $\boldsymbol{s}$. By $border(\boldsymbol{s})$ we denote the length of the longest border of $\boldsymbol{s}$ that is shorter than $\boldsymbol{s}$.

**Definition 3 (Border array)** *For a* ***string*** $\boldsymbol{s} \in \Sigma^n$*, the border array* $\beta[1\,..\,n]$ *is defined by* $\beta[i] = |border(\boldsymbol{s}[1\,..\,i])|$ *for* $1 \leq i \leq n$.

A border of a non-empty ***string*** $\boldsymbol{s}$ is a proper factor $\boldsymbol{u}$ that is both a prefix and a suffix of $\boldsymbol{s}$. The notions of period and of border are dual. It is a known fact [CHL07] that, for any non-empty ***string*** $\boldsymbol{s}$, it holds.

$$Period(\boldsymbol{s}) + Border(\boldsymbol{s}) = |\boldsymbol{s}| \tag{1.1}$$

### 1.1.2 Covers and Seeds

A substring $\boldsymbol{u}$ of $\boldsymbol{s}$ is called a cover of $\boldsymbol{s}$, if $\boldsymbol{s}$ can be constructed by concatenating or overlapping copies of $\boldsymbol{u}$.

We also say that $\boldsymbol{u}$ covers $\boldsymbol{s}$. For example, if $\boldsymbol{s} = ababaaba$, then $aba$ and $\boldsymbol{s}$ covers $\boldsymbol{s}$. If $\boldsymbol{s}$ has a cover $\boldsymbol{u} \neq \boldsymbol{s}$, $\boldsymbol{s}$ is said to be quasi-periodic; otherwise, $\boldsymbol{s}$ is super-primitive.

A substring $\boldsymbol{u}$ of $\boldsymbol{s}$ is called a seed of $\boldsymbol{s}$, if $\boldsymbol{u}$ covers one super-string of $\boldsymbol{s}$ (this can be any super-string of $\boldsymbol{s}$, including $\boldsymbol{s}$ itself). For example, $aba$ and $ababa$ are some seeds of $\boldsymbol{s} = ababaab$.

For set $\mathcal{G}$ of positive integers, we define the $maxgap$ of $\mathcal{G}$ as:

$$maxgap(\mathcal{G}) = \begin{cases} max\{\mathcal{G}[i+1] - \mathcal{G}[i] \text{: for } 1 \leq i < n\} \text{ if } |\mathcal{G}| > 1 \\ 0 \text{ otherwise} \end{cases}$$

By border(s) we denote the For a factor $\boldsymbol{u}$ of $\boldsymbol{s}$, by $Occ(\boldsymbol{u}, \boldsymbol{s})$ we denote the set of indices of starting positions of all occurrences of $\boldsymbol{u}$ in $\boldsymbol{s}$.

## 1.2 Strings similarity measurements

**Definition 4 (Edit Distance)** *[Gus97, Dam64, Lev66] The distance $\delta(\boldsymbol{x}, \boldsymbol{y})$ between two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is the minimal cost of a sequence of operations that transform $\boldsymbol{x}$ into $\boldsymbol{y}$ (and $\infty$ if no such sequence exists). The cost of a sequence of operations is the sum of the costs of the individual operations. The operations are a finite set of rules of the form $\delta(\boldsymbol{u}, \boldsymbol{v}) = n$, where $\boldsymbol{u}$ and $\boldsymbol{v}$ are different strings and $n$ is a non-negative real number. Once the operation has converted a **string** $\boldsymbol{u}$ into $\boldsymbol{v}$, no further operations can be done on $\boldsymbol{v}$.*

- *Insertion: $\delta(\epsilon, a)$, inserting the letter $a$.*
- *Deletion: $\delta(a, \epsilon)$, deleting the letter $a$.*
- *Substitution: $\delta(a, b)$ for $a \neq b$, substituting $a$ by $b$.*
- *Transposition: $\delta(ab, ba)$ for $a \neq b$, swap the adjacent letters $ab$ and $ba$.*

*Note that $\delta(\boldsymbol{x}, \boldsymbol{y})$ can be computed in $\mathcal{O}(|\boldsymbol{x}| \times |\boldsymbol{y}|)$ time using dynamic programming. using the equation (edit) score of best alignment from $x_1 \cdot\cdot x_i$ to $y_1 \cdot\cdot y_j$.*

$$\delta(i,j) = \begin{cases} \delta(i-1, j-1), \ if\ x_i = y_j & copy \\ \delta(i-1, j-1) + 1, \ if\ x_i \neq y_j & substitute \\ \delta(i-1, j) + 1 & insert \\ \delta(i, j-1) + 1 & delete \\ \delta(i, j-1) + 1 & transposition \end{cases} \tag{1.2}$$

**Definition 5 (Hamming Distance)** *Given two strings of equal length, the Hamming distance between them is the number of positions for which the corresponding symbols are different. In other words, the Hamming distance between two strings of equal length is the minimum number of symbol substitutions required to change one **string** into the other.*

*Hamming distance [SK83]: allows only substitutions, which cost $1$ in the simplified definition. In the literature the search problem in many cases is called "string matching*

*with $k$ mismatches". The distance is symmetric, and it is finite whenever $|\boldsymbol{x}| = |\boldsymbol{y}|$. In this case it holds $0 \leq \delta(\boldsymbol{x}, \boldsymbol{y}) \leq |\boldsymbol{x}|$*

$$\delta_H(\boldsymbol{x}, \boldsymbol{y}) = |I|, I = \{i | x_i \neq y_i, 1 \leq i \leq n\} \text{ where } |\boldsymbol{x}| = |\boldsymbol{y}| = n \tag{1.3}$$

**Definition 6 (Alignment)** *An alignment [CHL07] between two strings $\boldsymbol{x}, \boldsymbol{y} \in \Sigma^*$ whose respective lengths are $n$ and $m$, is a way to visualize their similarities, Formally an alignment $\mathcal{A}$ between $\boldsymbol{x}$ and $\boldsymbol{y}$ is a **string** $\boldsymbol{z}$ such that $(\Sigma \cup \epsilon)) \times (\Sigma \cup \epsilon) \times (\epsilon, \epsilon)$.*

*Given two sequences $\boldsymbol{x}$ and $\boldsymbol{y}$ such that $\boldsymbol{x} = \langle x_1, x_2, \cdot\cdot, x_n \rangle, \boldsymbol{y} = \langle y_1, y_2, \cdot\cdot, y_n \rangle$,. Formally an alignment score between $\boldsymbol{x}$ and $\boldsymbol{y}$ is $\mathcal{A}_{score}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{|\boldsymbol{x}'|} S(\boldsymbol{x}'_i \boldsymbol{y}'_i)$. The problem of finding the optimal global alignment becomes that of finding the alignment that maximizes the number of matches, assuming A score function $\delta(\boldsymbol{x}, \boldsymbol{y})$ is defined for each $(\boldsymbol{x}, \boldsymbol{y}) \in \Sigma \times \Sigma$.*

**Definition 7 (Exact pattern matching problem)** *String matching involves finding one, or more generally, all the occurrences of a **string** $\boldsymbol{p}$ (called a pattern) in a text $\boldsymbol{t}$. The text (a **string** of length $n$) is denoted by $\boldsymbol{t} = \boldsymbol{t}[1..n]$. The pattern (a **string** of length $m$) is denoted by $p = p[1\,..\,m]$. Both strings are drawn over a finite alphabet $\Sigma$.*

*Given a **string** $\boldsymbol{t}$ of length $n$ and a pattern $\boldsymbol{P}$ (a **string** of length $m$) both over an alphabet $\Sigma$, The occurrence positions of $\boldsymbol{p}$ in $\boldsymbol{t}$ are defined as:*

$$Occ = \{|\boldsymbol{u}| + 1,\ \exists\, \boldsymbol{u}, \boldsymbol{v} \in \Sigma^*;\ \boldsymbol{t} = \boldsymbol{upv}\}$$

- $\boldsymbol{exists}(\boldsymbol{p},\ \boldsymbol{t})$ *returns true $iff$ $\boldsymbol{p}$ is in $\boldsymbol{t}$, i.e., returns true $iff |Occ(\boldsymbol{p}, \boldsymbol{t})| > 0$*
- $\boldsymbol{count}(\boldsymbol{p},\ \boldsymbol{t})$ *counts the number of occurrences of $\boldsymbol{p}$ in $\boldsymbol{t}$, i.e., returns $|Occ(\boldsymbol{p}, \boldsymbol{t})|$.*
- $\boldsymbol{locate}(\boldsymbol{p},\ \boldsymbol{t})$ *finds all the occurrences of $\boldsymbol{p}$ in $\boldsymbol{t}$, i.e., returns the ordered set $Occ(\boldsymbol{p}, \boldsymbol{t})$.*
- $\boldsymbol{extract}(\boldsymbol{t},\ i,\ j)$ *extracts the substring $\boldsymbol{t}[i..j]$.*

We use basic notions from [HU79], which we briefly discuss below. A Finite State Machine, alternatively, a Finite State Automaton (abbreviated as FSM or FSA respectively) is an abstract mathematical concept of computation that is capable of storing a status or state and changing the current state based on the input sequences.

We are interested in *deterministic* finite state machines. The term "deterministic" refers to the fact that on each input there is one and only one state to which the automaton can transition from its current state (as opposite to *non-deterministic* finite automaton, where there can be several states to transition to).

**Definition 8 (Finite State Automaton)** *A deterministic finite state automaton $A$ is defined as a five-tuple: $A = (Q, \Sigma, \delta, s, F)$, where $Q$ is a finite set of states, $\Sigma$ is a finite set of characters (the alphabet), $s \in Q$ is the initial state (element of $Q$), $F \subseteq Q$ is a set of accepting states (subset of $Q$), $\delta$ is a transition (mapping) function $\delta : Q \times \Sigma \to Q$ that takes as arguments a state and an input character and returns a state.*

For each $q \in Q$, $a \in A$ such that $\delta(p, a) = q$, we call $(p, a, q)$, also denoted by $p \xrightarrow{a} q$, an edge or arc of $A$. In the transition diagram, $\delta$ will be represented by edges/arcs between states with input character(s) as the labels on the edges. To elaborate, if $p$ is a state, $\alpha$ is an input character, and $\delta(p, \alpha)$ is the state $q$ then there exists an edge labeled $\alpha$ from $p$ to $q$. A path is defined as a sequence of consecutive edges. A path is accepted if its ending state is a final state. The sequence spelling the labels of the edges constituting the path is then also assumed to be accepted. Let $\mathcal{L}$ be a finite language and $A$ is a DFA that accepts all words in $L$, so a word $\boldsymbol{w}$ is in $\mathcal{L}$ if and only if it is accepted by $A$.

Informally, A Finite State Transducer (FST) is a special type of finite state automaton that works on two (or more) tapes. Rather than just traversing (and accepting or rejecting) an input string, a transducer works like a sort of "translating machine". It reads from one of the tapes and write onto the other. In the *translation mode*, an FST translates the contents of its input ***string*** to its output string. In the *generation mode*, it accepts a ***string*** on its input tape and generates another ***string*** on its output tape.

**Definition 9 (Finite State Transducer (FST))** *Formally, a FST $T$ is a 6-tuple $T = (Q, \Sigma, \Gamma, I, F, \delta)$ where $Q$ is a finite set of states, $\Sigma$ is a finite set of characters, called the input alphabet, $\Gamma$ is a finite set of characters, called the output alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and finally $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the transition relation, i.e., $\delta$ is a mapping function from $Q \times \Sigma$ to finite subsets of $Q \times \Gamma$. Here, $\epsilon$ is the empty string.*

## 1.3 Fundamental Data structures

### 1.3.1 Graph

A graph is an abstraction used to model a system that contains discrete, interconnected elements. The elements are represented by nodes (also called vertices) and the interconnections are represented by edges.

**Definition 10 (Graph)** *Formally, a graph $G$ is an ordered triple of sets $(V, E, \Psi)$, where:*

- *$V$ is a nonempty set whose elements are called vertices.*
- *$E$ is a collection of pair subsets of $V$ called edges.*
- *$\Psi : E \to \mathbb{R}$ is a function associated with each edge of $G$*

*Two vertices are adjacent when they are both incident to a common edge. A path is a sequence of vertices $P = (v_1, v_2, \ldots, v_n)$ in a graph such that $v_i$ and $v_{i+1}$ are adjacent for $i \leq i \leq n$. The length of the path is the number of edges traversed.*

#### BREADTH-FIRST SEARCH

In graph theory, breadth-first search (BFS), it derives its name from the way it expands a "search front" from a start point (in the decision tree, as opposed to depth-first search (DFS)), is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. The algorithm belongs to a major class of graph search algorithms based on a technique called "relaxation".

The shortest path from $v_1 \rightsquigarrow v_n$ is the path $P = (v_1, v_2, \ldots, v_n)$ that over all possible $n$ minimizes the sum $\sum_{i=1}^{n} \Psi(e_{i,i+1})$. When each edge in the graph has unit weight or $f : E \to \{1\}$, this is equivalent to finding the path with fewest edges.

That is, the graph is explored level by level. In fact, all shortest paths of one arc are first computed, followed by those made up of two arcs, and so on.

Finding the shortest path between two nodes $u$ and $v$ using breadth-first search (BFS) is obtained by processing edges using a *queue* data structure . It processes the

vertices in the graph in the order of their shortest distance from the vertex $v_s$ (the start vertex).

A *queue* is a list of elements which supports the following operations

- enqueue: Adds an element to the end of the list
- dequeue: Removes an element from the front of the list

Elements are extracted in first-in first-out (FIFO) order, i.e., elements are picked in the order in which they were inserted.

### 1.3.2 Suffix tree

The suffix tree is a fundamental data structure [Gus97] supporting a wide variety of efficient ***string*** processing algorithms. The suffix tree of ***string*** $\boldsymbol{s}$, denoted $ST(\boldsymbol{s})$ or simply $ST$, is a compacted trie of all suffixes of ***string*** $\boldsymbol{s}$. Let $|\boldsymbol{s}| = n$. It has the following properties:

1. The tree has $n$ leaves, labelled $1, 2, \ldots, n$, one corresponding to each suffix of $\boldsymbol{s}$.
2. Each internal node has at least 2 children.
3. Each edge in the tree is labelled with a substring of $\boldsymbol{s}$.
4. The concatenation of edge labels from the root to the leaf labelled $i$ is $suff(i)$.
5. The labels of the edges connecting a node with its children start with different characters.

The Suffix tree is a data structure that admits efficient on-line ***string*** searches.

Weiner [Wei73] was the first to show that suffix trees can be built in linear time (Knuth is claimed to have called it ”the algorithm of 1973”). More space efficient algorithm was introduced by McCreight in 1976 [McC76].

Esko Ukkonen [Ukk95] devised a linear-time algorithm for constricting a suffix tree that may be the conceptually easiest linear-time construction algorithm. However, Ukkonen’s method is equally fast and uses far less space (i.e., memory) in practice than Weiner’s method

Ukkonen's idea [Ukk95] for computing suffix trees in linear time is a combination of several nice insights into the structure of suffix trees and several clever implementation details - Hence Ukkonen is the method of choice for most problems requiring the construction of a suffix tree.

Suffix trees permit on-line ***string*** searches of the type $\boldsymbol{exists}$, to be answered in $\mathcal{O}(|\boldsymbol{p}| \log |\Sigma|)$ time, where $|\boldsymbol{p}|$ is the length pattern. We explicitly consider the dependence of the complexity of the algorithms on $|\Sigma|$, rather than assume that it is a fixed constant, because can be quite large for many applications. A suffix tree is able to find all the occurrences $Occ$ of a pattern $\boldsymbol{p}$ of length $m$ in time $\mathcal{O}(m + |Occ|)$. Suffix trees can also be constructed in time $\mathcal{O}(n)$ with $\mathcal{O}(|\boldsymbol{p}|)$ time for a query, but this requires $\mathcal{O}(n \times |\Sigma|)$ space, which renders this method impractical in many applications.

### 1.3.3 Suffix array

In 1990 Manber and Myers introduced suffix arrays [MM90] as a space-saving alternative to suffix trees and described original algorithm for suffix array construction and use.

The suffix array of a ***string*** $\boldsymbol{s}$ of length $|\boldsymbol{s}| = n$ is a lexicographically ordered array of the set $\boldsymbol{s}[i\,..\,n]$, where $0 \leq i \leq n$, of all suffixes of $\boldsymbol{s}$. As with suffix trees, it is common to add the end symbol $s[n] = \$$. It has no effect on the suffix array assuming \$ is smaller than any other symbol.

More precisely, the suffix array is an array $SA[0\,..\,n]$ of integers containing a permutation of the set $\boldsymbol{s}[i\,..\,n]$, where $0 \leq i \leq n$, such that

$$SA(\boldsymbol{s})[0] < SA(\boldsymbol{s})[1] < .. < SA(\boldsymbol{s})[n]$$

Suffix array is much simpler data structure than suffix tree (see also [Gus97, PST07, PST06]). In particular, the type and the size of the alphabet are usually not a concern. Therefor, the size on the suffix array is linear on any alphabet, also the suffix array can be constructed in the same asymptotic time required to sort the characters of the given string [KS03, KA05].

Suffix array construction algorithms are quite fast in practice too. For example, the fastest way to construct a suffix tree is to construct a suffix array first and then use it to construct the suffix tree.

Suffix arrays are rarely used alone but are augmented with other arrays and data structures depending on the application. In particular, the *longest common prefix (LCP) array*. It stores the lengths of the longest common prefixes between pairs of consecutive suffixes in the suffix array.

**Definition 11 (Longest Common Prefix array (LCP array))** *is an auxiliary data structure to the suffix array.*

$$LCP[i](\boldsymbol{s}) = lcp(SA[i](\boldsymbol{s}), SA[i-1](\boldsymbol{s})), \forall i \in [1\,..\,n]$$

LCP *can be computed in linear time using the suffix array and its inverse.*

## 1.4 Problems as Languages

In complexity theory, problem solving can be viewed as language recognition. A problem is expressed in the form of a question and has some parameter(s). Each set of parameter(s) makes a new instance of the problem. It can be thought of as the general problem as a template where parameters are indicated by symbols (for the sake of description) and of an instance as replacing the symbolic parameters with actual values.

Based on the type of expected solution, problems could be classified in to the following general categories:

**Optimization problem** is the selection of an optimal element (with regard to some criteria) from some set of available alternatives. Usually algorithms have an optimization goal in mind, e.g. compute the shortest path or the alignment or minimal edit distance

**Decision problem** where the question statement of a problem asks whether a solution with specific characteristics exists or not. In other words a problem is stated as a yes or no question depending on the values of some input parameters

Another important distinction between types of algorithms is, whether the algorithm belongs to deterministic or randomized group. Deterministic algorithms always produce the same results on a given input following the same computation steps while

Randomized algorithms (also called Monte Carlo algorithms) roll dice during execution. Hence either the order of execution or the result of the algorithm might be different for each run on the same input.

Problems have to be formalized in order to be tackled systematically by computational algorithms. Therefore, all the problems have to be first formally defined where the given instance and question are stated clearly.

These algorithmic problems are then classified according to their computational complexity where problems having the same computational complexity (typically defined in terms of space and time resources required to solve the problem) are said to belong to the same complexity class.

By knowing the performance of an algorithm under each of these cases, you can judge whether an algorithm is appropriate for use in your specific situation.

### 1.4.1 The Computational Complexity of a Problem

Algorithms are compared by evaluating their performance and how they respond (e.g., in their processing time or working space requirements) to changes in input data size.

In particular, the order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms.

This methodology is the standard means developed over the past half-century for comparing algorithms called "asymptotic notation" or Bachmann - Landau notation (after Edmund Landau and Paul Bachmann). By doing so, we can determine which algorithms scale to solve problems of a non-trivial size by evaluating the running time and space memory needed by the algorithm in relation to the size of the provided input.

The running time of an algorithm is defined as the number of steps that the algorithm requires to solve the problem.

Basic operations such as single addition, subtraction or multiplication are considered to take unit time. The notion of algorithm run-time adopted in complexity theory does not regard the type of machine used or time elapsed in usual units of minutes and seconds, the assumption here is that the run-time depends only on the algorithm and the input.

The following definitions from [CLRS09].

Assume two functions $f$ and $g$, such that $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$.

**Definition 12** ($\Theta-notation$) *The $\theta$-notation asymptotically bounds a function from above and below. For a given function $g(n)$, we denote by $theta(g(n))$ the set of functions.*

$$\theta(g(n)) = \{f(n) : \textit{ there exist a positive constants } c_1, c_2 \textit{ and } n_0, \textit{ such that } 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n) \forall n \geq n_0\}.$$

**Definition 13** ($\mathcal{O}-notation$) *For a given function $g(n)$, we denote by $\mathcal{O}(g(n))$ (pronounced "big-oh of $g$ of $n$" or sometimes just "oh of $g$ of $n$") the set of functions*

$$\mathcal{O}(g(n)) = \{f(n) : \textit{ there exist a positive constants } c \textit{ and } n_0, \textit{ such that } 0 \leq f(n) \leq c\, g(n) \forall n \geq n_0\}.$$

**Definition 14** ($\Omega-notation$) *Just as $\mathcal{O}$-notation provides an asymptotic upper bound on a function, $\Omega$-notation provides an asymptotic lower bound. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced @big-omega of $g$ of $n$" or sometimes just "omega of $g$ of $n$") the set of functions*

$$\Omega(g(n)) = \{f(n) : \textit{ there exist a positive constants } c \textit{ and } n_0, \textit{ such that } 0 \leq c\, g(n) \leq f(n) \forall n \geq n_0\}.$$

Finally, we list some commonly used adjectives describing classes of functions. We say a function $f$ is:

- Constant, **if** $f(n) \in \theta(1)$.
- Logarithmic, **if** $f(n) \in \mathcal{O}(\log n)$.
- Poly - Logarithmic, **if** $f(n) \in \mathcal{O}(n \times \log^k n), \forall k \in \mathbb{N}$.
- Linear, **if** $f(n) \in \mathcal{O}(n)$.
- Quadratic, **if** $f(n) \in \mathcal{O}(n^2)$.
- Exponential, **if** $f(n) \in \mathcal{O}(2^n)$.

# Chapter II

# Structured regularities on strings

# Article: # 1

# Inferring an Indeterminate String from a Prefix Graph

An ***indeterminate string*** (or, more simply, just a ***string***) $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet $\Sigma$ is a sequence of nonempty subsets of $\Sigma$. We say that $\boldsymbol{x}[i_1]$ and $\boldsymbol{x}[i_2]$ ***match*** (written $\boldsymbol{x}[i_1] \approx \boldsymbol{x}[i_2]$) if and only if $\boldsymbol{x}[i_1] \cap \boldsymbol{x}[i_2] \neq \emptyset$. A ***feasible array*** is an array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ of integers such that $\boldsymbol{y}[1] = n$ and for every $i \in 2..n$, $\boldsymbol{y}[i] \in 0..n-i+1$. A ***prefix table*** of a string $\boldsymbol{x}$ is an array $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$ of integers such that, for every $i \in 1..n$, $\boldsymbol{\pi}[i] = j$ if and only if $\boldsymbol{x}[i..i+j-1]$ is the longest substring at position $i$ of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. It is known from [CRSW14] that every feasible array is a prefix table of some indetermintate string. A ***prefix graph*** $\mathcal{P} = \mathcal{P}_{\boldsymbol{y}}$ is a labelled simple graph whose structure is determined by a feasible array $\boldsymbol{y}$. In this article, we show, given a feasible array $\boldsymbol{y}$, how to use $\mathcal{P}_{\boldsymbol{y}}$ to construct a lexicographically least indeterminate string on a minimum alphabet whose prefix table $\boldsymbol{\pi} = \boldsymbol{y}$.

---

## 1.1 Introduction

In the extensive literature of stringology/combinatorics on words, a "string" or "word" has usually been defined as a sequence of individual elements of a distinguished set $\Sigma$ called an "alphabet". Nevertheless, going back as far as the groundbreaking paper of Fischer & Paterson [FP74], more general sequences, defined instead on *subsets* of $\Sigma$, have also been considered. The more constrained model introduced in [FP74] restricts entries in a string to be either elements of $\Sigma$ (subsets of size 1) or $\Sigma$ itself (subsets of size $\sigma = |\Sigma|$); these have been studied in recent years as "strings with don't cares" [IMM+03], also "strings with holes" or "partial words" [BS08]. The unconstrained model, which allows arbitrary nonempty subsets of $\Sigma$, has also attracted significant attention, often because of applications in bioinformatics: such strings have variously been called "generalized" [Abr87], "indeterminate" [HS03], or "degenerate" [IMR08].

In this article, we study strings in their full generality, hence the following definitions:

**Definition 15** *Suppose a set $\Sigma$ of symbols (called the **alphabet**) is given. A **string** $\boldsymbol{x}$ on $\Sigma$ of **length** $n = |\boldsymbol{x}|$ is a sequence of $n \geq 0$ nonempty finite subsets of $\Sigma$, called **letters**; we represent $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$. If $n = 0$, $\boldsymbol{x}$ is called the **empty string** and denoted by $\varepsilon$; if for every $i \in 1..n$, $\boldsymbol{x}[i]$ is a subset of $\Sigma$ of size 1, $\boldsymbol{x}$ is said to be a **regular string**.*

**Definition 16** *Suppose we are given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ and integers $i \in 1..|\boldsymbol{x}|$, $j \in 1..|\boldsymbol{y}|$. We say that $\boldsymbol{x}[i]$ and $\boldsymbol{y}[j]$ **match** (written $\boldsymbol{x}[i] \approx \boldsymbol{y}[j]$) if and only if $\boldsymbol{x}[i] \cap \boldsymbol{y}[j] \neq \emptyset$. Then $\boldsymbol{x}$ and $\boldsymbol{y}$ **match** ($\boldsymbol{x} \approx \boldsymbol{y}$) if and only if $|\boldsymbol{x}| = |\boldsymbol{y}|$ and $\boldsymbol{x}[i] \approx \boldsymbol{y}[i]$ for every $i \in 1..|\boldsymbol{x}|$.*

Note that matching is not necessarily transitive: $a \approx \{a, b\} \approx b$, but $a \not\approx b$.

**Definition 17** *The **prefix table** (also **prefix array**)[1] of a string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is the integer array $\boldsymbol{\pi_x} = \boldsymbol{\pi_x}[1..n]$ such that for every $i \in 1..n$, $\boldsymbol{\pi_x}[i]$ is the length of the longest prefix of $\boldsymbol{x}[i..n]$ that matches a prefix of $\boldsymbol{x}$. Thus for every prefix table $\boldsymbol{\pi_x}$, $\boldsymbol{\pi_x}[1] = n$. When there is no ambiguity, we write $\boldsymbol{\pi} = \boldsymbol{\pi_x}$.*

[1] We prefer "table" because of the possible confusion with "suffix array", a completely different data structure.

The prefix table is an important data structure for strings: it identifies all the borders, hence all the periods, of every prefix of $\boldsymbol{x}$ [CRSW14]. It was originally introduced to facilitate the computation of repetitions in regular strings [ML84], see also [Smy03]; and for regular strings, prefix table and border array are equivalent, since each can be computed from the other in linear time [BKS13]. For general strings, the prefix table can be computed in compressed form in $O(n^2)$ time using $\Theta(n/\sigma)$ bytes of storage space [SW08], where $\sigma = |\Sigma|$. Two examples follow, adapted from [CRSW14]:

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x_1} = & a & c & a & g & a & c & a & t \\ \boldsymbol{\pi_1} = & 8 & 0 & 1 & 0 & 3 & 0 & 1 & 0 \end{array} \tag{1.1}$$

$$\begin{array}{rcccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \boldsymbol{x_2} = & \{a,c\} & \{g,t\} & \{a,g\} & \{a,c,g\} & g & c & \{a,t\} & a \\ \boldsymbol{\pi_2} = & 8 & 0 & 4 & 2 & 0 & 3 & 1 & 1 \end{array} \tag{1.2}$$

Since clearly every position $i \in 2..n$ in a prefix table $\boldsymbol{\pi}$ must satisfy $0 \leq \boldsymbol{\pi}[i] \leq n-i+1$, the following definition is a natural one:

> An array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ of integers is said to be a ***feasible array*** if and only if $\boldsymbol{y}[1] = n$ and for every $i \in 2..n$, $\boldsymbol{y}[i] \in 0..n-i+1$.

An immediate consequence of Definition 17 is the following:

**Lemma 1.1.1 ([CRSW14])** *Let $\boldsymbol{x} = \boldsymbol{x}[1..n]$ be a string. An integer array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ is the prefix table of $\boldsymbol{x}$ if and only if for each position $i \in 1..n$, the following two conditions hold:*

*(a)* $\boldsymbol{x}\big[1..\boldsymbol{y}[i]\big] \approx \boldsymbol{x}\big[i..i+\boldsymbol{y}[i]-1\big]$;

*(b) if* $i + \boldsymbol{y}[i] \leq n$, *then* $\boldsymbol{x}\big[y[i]+1\big] \not\approx \boldsymbol{x}\big[i+\boldsymbol{y}[i]\big]$.

Then the following fundamental result establishes the important connection between strings and feasible arrays:

**Lemma 1 ([CRSW14])** *Every feasible array is the prefix table of some string.*

In view of this lemma, we say that a feasible array is ***regular*** if it is the prefix array of a regular string. We are now able to state the goal of this article as follows: for a given feasible array $\boldsymbol{y} = \boldsymbol{y}[1..n]$, not necessarily regular, construct a string $\boldsymbol{x}$ on a minimum alphabet whose prefix table $\boldsymbol{\pi_x} = \boldsymbol{y}$ — the "reverse engineering" problem for the prefix table in its full generality. In fact, we do somewhat more: we construct a lexicographically least such string, in a sense to be defined in Section 1.2.

The first reverse engineering problem in stringology was stated and solved in [FLR+99, FGL+00], where an algorithm was described to compute a lexicographically least regular string whose border array was a given integer array — or to return the result that no such regular string exists. Many other similar constructions have since been published, related to other stringological data structures but always specific to regular strings (see [BIST03, DLL05, FS06, MNRR13], among others). [NRR12] was the first paper to consider the more general problem of inferring an indeterminate string from a given data structure (specifically, border array, suffix array and LCP array). Although solving such problems does not yield immediate applications, nevertheless solutions provide a deeper understanding of the combinatorial many-many relationship between strings and the various data structures developed from them (see for example [MSM99], where canonical strings corresponding to given border arrays are identified and efficiently generated for use as test data).

For prefix tables and regular strings, the reverse engineering problem was solved in [CCR09], where a linear-time algorithm was described to return a lexicographically least regular string $\boldsymbol{x}$ whose prefix table is the given feasible array $\boldsymbol{y}$, or an error message if no such $\boldsymbol{x}$ exists. A recent paper [BSBW14] sketches two algorithms to compute an indeterminate string $\boldsymbol{x}$ on a minimum alphabet (not necessarily lexicographically least) corresponding to a given feasible array $\boldsymbol{y}$, but the algorithms are theoretical in nature: one requires the determination of the chromatic number of a certain graph, an NP-hard problem, while the other depends on somehow identifying the minimum "induced positive edge cover" of a graph. However, [BSBW14] proves an important result that we use below to bound the complexity of our algorithm: that the minimum alphabet size $\sigma$ of a string corresponding to a given feasible array of length $n$ is at most $n+\sqrt{n}$. In this article, we use graph-theoretic methods developed from [CRSW14] to compute a lexicographically least string, regular or not, corresponding to the given $\boldsymbol{y}$, in time $O(\sigma n^2)$.

Section 1.2 of this article provides background material for an understanding of our algorithm; Section 1.3 presents the algorithm itself; Section 1.4 briefly discusses these results and suggests future work.

## 1.2 Preliminaries

Following [CRSW14], for a given feasible array $\boldsymbol{y} = \boldsymbol{y}[1..n]$, we define a corresponding graph $\mathcal{P} = \mathcal{P}_{\boldsymbol{y}}$, on which our algorithm will be based:

**Definition 18** *Let $\mathcal{P} = (V, E)$ be a labelled graph with vertex set $V = \{1, 2, \ldots, n\}$ consisting of positions in a given feasible array $\boldsymbol{y}$. In $\mathcal{P}$ we define, for $i \in 2..n$, two kinds of edge (compare Lemma 1.1.1):*

*(a) for every $h \in 1..\boldsymbol{y}[i]$, $(h, i+h-1)$ is called a **positive edge**;*

*(b) $(1+\boldsymbol{y}[i], i+\boldsymbol{y}[i])$ is called a **negative edge**, provided $i+\boldsymbol{y}[i] \leq n$.*

*$E^+$ and $E^-$ denote the sets of positive and negative edges, respectively. We write $E = E^+ \cup E^-$, $\mathcal{P}^+ = (V, E^+)$, $\mathcal{P}^- = (V, E^-)$, and we call $\mathcal{P}$ the **prefix graph** $\mathcal{P}_{\boldsymbol{y}}$ of $\boldsymbol{y}$. If $\boldsymbol{x}$ is a string having $\boldsymbol{y}$ as its prefix table, then we also refer to $\mathcal{P}$ as the **prefix graph** $\mathcal{P}_{\boldsymbol{x}}$ of $\boldsymbol{x}$.*

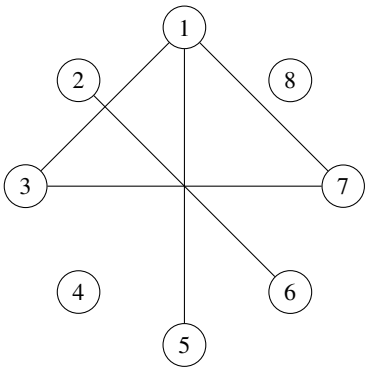



Figure 1.1: $\mathcal{P}^+_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 80103010$

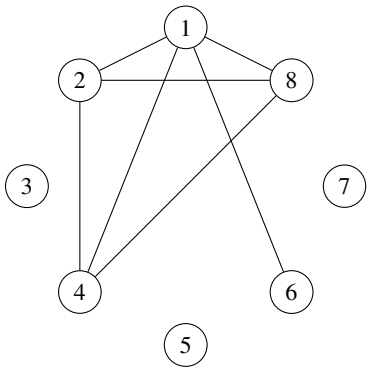



Figure 1.2: $\mathcal{P}^-_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 80103010$

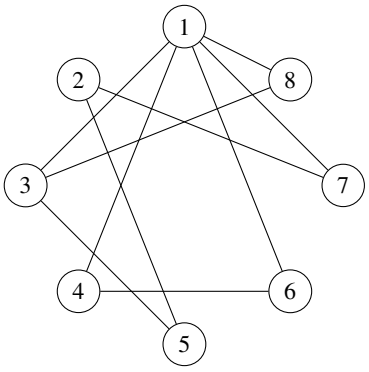



Figure 1.3: $\mathcal{P}^{+}_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 80420311$

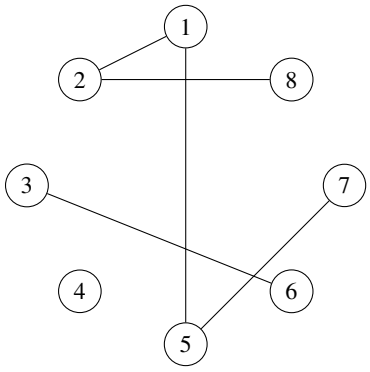



Figure 1.4: $\mathcal{P}^{-}_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 80420311$

Observe that $E^+$ and $E^-$ are necessarily disjoint. Figures 1.1–1.4 show the prefix graphs, as given in [CRSW14], for the example strings (1.1) and (1.2). Again, in Figures 1.5–1.8 we present the prefix graphs of two different indeterminate strings (1.3) and (1.4) given below.

From Definition 18, we see that $|E^+|$ can be as small as 0 (for example, when $\boldsymbol{s} = ab^{n-1}$) or as large as $\binom{2}{n}$ (where $\boldsymbol{s} = a^n$). We see that every edge $(i, j) \in E^-$ determines the value $\boldsymbol{y}[ji + 1]$ of a position $ji + 1$ in $\boldsymbol{y}$. Thus a simple scan of $y$ can identify all positions $h$ that are ***not*** determined by $E^-$; for all such $h$, it must be true that $\boldsymbol{y}[h] = nh + 1$.

**Remark 1.2.1 ([CRSW14])** *The prefix array and the negative prefix graph provide the same information and so determine the same set of (regular or not) strings* $\boldsymbol{s}$.

$$\begin{array}{lcccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} \\ \boldsymbol{x_3} = & \{a,b\} & \{a,c\} & c & \{a,b\} & b & c & \{a,c\} & b \\ \boldsymbol{\pi_3} = & 8 & 2 & 0 & 1 & 4 & 0 & 1 & 1 \end{array} \tag{1.3}$$

$$\begin{array}{lcccccccc} & {\scriptstyle 1} & {\scriptstyle 2} & {\scriptstyle 3} & {\scriptstyle 4} & {\scriptstyle 5} & {\scriptstyle 6} & {\scriptstyle 7} & {\scriptstyle 8} \\ \boldsymbol{x_4} = & \{a,b\} & \{a,c\} & \{a,d\} & \{c,e\} & a & \{b,e\} & c & d \\ \boldsymbol{\pi_4} = & 8 & 2 & 4 & 0 & 1 & 3 & 0 & 0 \end{array} \tag{1.4}$$

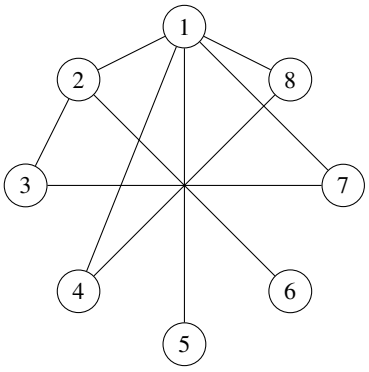



Figure 1.5: $\mathcal{P}^{+}_{\boldsymbol{y_3}}$ for $\boldsymbol{y_3} = 82014011$

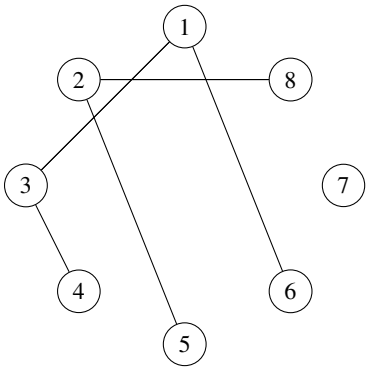



Figure 1.6: $\mathcal{P}^{-}_{\boldsymbol{y_3}}$ for $\boldsymbol{y_3} = 82014011$

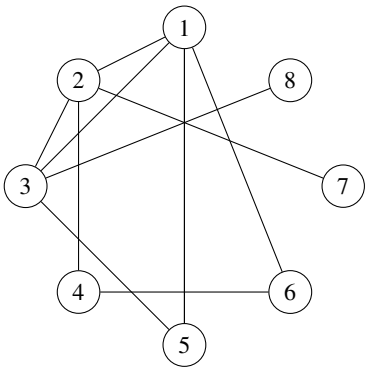



Figure 1.7: $\mathcal{P}^{+}_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 82401300$

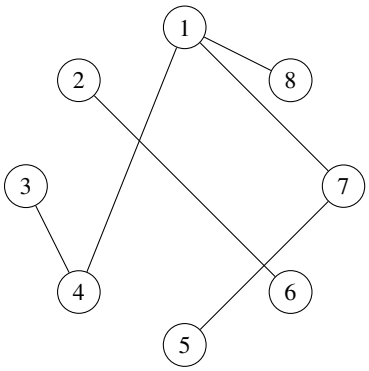



Figure 1.8: $\mathcal{P}^{-}_{\boldsymbol{y_4}}$ for $\boldsymbol{y_4} = 82401300$

In what follows the following notion of a regular table (or regular array) will be useful. In particular, a feasible array that is a prefix array of a regular string is said to be regular. The following lemma characterizes a regular table/array and will be useful for the analysis of our algorithm:

**Lemma 2 ([CRSW14])** *Let $\mathcal{P}_{\boldsymbol{y}} = (V, E)$ be a prefix graph of a feasible array $\boldsymbol{y}$. Then $\boldsymbol{y}$ is regular if and only if every edge of $P^{-}_{\boldsymbol{y}}$ joins two vertices in disjoint connected components of $P^{+}_{\boldsymbol{y}}$.*

So as to discuss the lexicographical ordering of strings on an ordered alphabet $\Sigma$, we need first of all a definition of the order of two letters:

**Definition 19** *Suppose two letters $\lambda$ and $\mu$ are given, where*

$$\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_j\},\ \mu = \{\mu_1, \mu_2, \ldots, \mu_k\},$$

*with $\lambda_h \in \Sigma$ for every $h \in 1..j$, $\mu_h \in \Sigma$ for every $h \in 1..k$. We assume without loss of generality that $j \leq k$, also that $\lambda_h < \lambda_{h+1}$ for every $h \in 1..j-1$ and $\mu_h < \mu_{h+1}$ for every $h \in 1..k-1$. Then $\lambda = \mu$ if and only if $\lambda_h = \mu_h$ for every $h \in 1..k$ and $j = k$; while $\lambda \prec \mu$ if and only if*

*(a) $\lambda_h = \mu_h$ for every $h \in 1..j < k$; or*

*(b) $\lambda_h = \mu_h$ for every $h \in 1..h' < j$* **and** *$\lambda_{h'+1} < \mu_{h'+1}$.*

*Otherwise, $\mu \prec \lambda$.*

Note that $(\lambda = \mu) \Rightarrow (\lambda \approx \mu)$, but that $\lambda \approx \mu$ implies neither equality nor an ordering of $\lambda$ and $\mu$. We remark also that the definition of letter order given here is not the only possible or useful one. For example, it would require $\{a, b, w, x, y, z\} \prec \{a, c\}$, thus arguably not placing sufficient emphasis on economy of letter selection in the alphabet.

**Definition 20** *Now suppose that two strings $\boldsymbol{x_1} = \boldsymbol{x_1}[1..n_1]$ and $\boldsymbol{x_2} = \boldsymbol{x_2}[1..n_2]$ on $\Sigma$ are given, where without loss of generality we assume that $n_1 \leq n_2$. Then $\boldsymbol{x_1} = \boldsymbol{x_2}$ if and only if $\boldsymbol{x_1}[h] = \boldsymbol{x_2}[h]$ for all $h \in 1..n_2$ and $n_1 = n_2$; while $\boldsymbol{x_1} \prec \boldsymbol{x_2}$ if and only if*

*(a) $\boldsymbol{x_1}[h] = \boldsymbol{x_2}[h]$ for every $h \in 1..n_1 < n_2$; or*

*(b) $\boldsymbol{x_1}[h] = \boldsymbol{x_2}[h]$ for every $h \in 1..h' < n_1$* **and** *$\boldsymbol{x_1}[h'+1] \prec \boldsymbol{x_2}[h'+1]$.*

*Otherwise, $\boldsymbol{x_2} \prec \boldsymbol{x_1}$.*

To better illustrate the relation of strings defined and used in this article we present the following examples:

- $\boldsymbol{x_1} = \{a, c\}\ \{g, t\}\ a \prec \boldsymbol{x_2} = \{a, c\}\ \{g, t\}\ \{a, g\}$
- $\boldsymbol{x_1} = a\ \{g, t\}\ \{a, c\}\ \{a, c, g\} \prec \boldsymbol{x_2} = a\ \{g, t\}\ \{a, t\}\ a$
- $\boldsymbol{x_1} = a\ \{a, c, g\}\ \{a, c, g\}\ \{a, t\} \prec \boldsymbol{x_2} =\ \{a, c\}\ g\ g\ \{a, t\}$

where $a < c < g < t$.

## 1.3 Algorithm REVENG

### 1.3.1 The Algorithm

The basic strategy of Algorithm REVENG, that constructs a lexicographically least string $\boldsymbol{x}$ (initially empty) corresponding to a given feasible array $\boldsymbol{y} = \boldsymbol{y}[1..n]$, is expressed by the main steps given below. Initially the alphabet $\Sigma$ is empty ($\sigma = 0$), as are the sets $\boldsymbol{x}[i]$, $1 \leq i \leq n$.

**(S1)** Consider the edges $(i, j)$ of $E^+$ in increasing order of $ni+j$ in order to add a single letter to $\boldsymbol{x}[i]$, $\boldsymbol{x}[j]$, or both based on the following steps;

**(S2)** if, by virtue of previous assignments, $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$ (so neither is empty), there is nothing to do — $(i, j)$ can be skipped;

**(S3)** otherwise, for the current $(i, j)$, determine a sequence

$$C = (\lambda_1, i_1), (\lambda_2, i_2), \ldots, (\lambda_r, i_r)$$

of all candidate assignments, where for every $h \in 1..r$, $i_h = i$ (respectively, $j$) if $\lambda_h \in \boldsymbol{x}[j]$ (respectively, $\boldsymbol{x}[i]$), and $\lambda_1 < \lambda_2 < \cdots < \lambda_r$;

**(S4)** for the current $h$, determine whether or not the assignment

$$\boldsymbol{x}[i_h] \leftarrow \boldsymbol{x}[i_h] \cup \{\lambda_h\}$$

is "allowable" (that is, compatible with the neighbourhood of $i_h$ in $E^-$) (detailed illustration to follow in (DS4)) — if so, then perform the assignment, maintaining the elements of $\boldsymbol{x}[i_h]$ in their natural order;

**(S5)** if for no $h$ is the assignment allowable, then assign a least new letter (drawn WLOG from the set of positive integers) to both $\boldsymbol{x}[i]$ and $\boldsymbol{x}[j]$;

**(S6)** since it may be that after Steps (S1)-(S5) have been executed for every $(i, j) \in E^+$, there still remain unassigned positions in $\boldsymbol{x}$ (that is, corresponding to isolated vertices in $\mathcal{P}^+$), a final assignment of a least possible letter for these positions is required (see function $\texttt{LEAST}(\texttt{i}, \lambda_{\max})$ and Lemma 1.3.1).

In order to implement this algorithm, several data structures need to be created, maintained, and accessed:

(DS1) The edges of $E^+$ are made accessible in increasing order for Step (S1) by a radix sort of the positive edges $(i, j)$ into a linked list $L^+$ (i.e., the linked list $L^+$ contains the edges of $E^+$ in increasing order), whose entries occur in increasing order of $i$ and, within each $i$, in increasing order of $j$. The time requirement is $\Theta(|E^+|)$, thus $O(n^2)$ in the worst case, since $E^+$ can contain $\Theta(n^2)$ edges [CRSW14].

(DS2) In order to implement Step (S4) of Algorithm REVENG, we need, for each position $i \in 1..n$, to have available a linked list of positions $j$ such that $(i, j)$ is an edge of $E^-$. This can be done by using $E^-$ to form a set of negative edges that includes each $(i, j)$ twice, both as $(i, j)$ and as $(j, i)$. Then in a preprocessing step the entries in this set are radix sorted into an array $N^- = N^-[1..n]$ of $n$ linked lists, such that for every $i \in 1..n$, $N^-[i]$ gives in increasing order all the vertices $j$ for which $(i, j) \in E^-$ (in other words, the neighbourhood of $i$ in $E^-$). Since $E^-$ contains $O(n)$ edges [CRSW14], this preprocessing step can be accomplished in $O(n)$ time.

(DS3) Steps (S2)-(S5) require that for each $i \in 1..n$, a linked list $\boldsymbol{x}[i]$ be maintained of letters $\lambda$ that have so far been assigned to $\boldsymbol{x}[i]$. Each list is maintained in increasing letter order, so that update, intersection, and union each require $O(\sigma)$ time, where $\sigma = |\Sigma|$ is the (current) alphabet size. Since for regular strings each $\boldsymbol{x}[i]$ has exactly one element, in this case processing time reduces to $O(1)$.

(DS4) In Step (S4), in order to determine whether a proposed assignment of a letter $\lambda_h$ to a position $i'_h$ in $\boldsymbol{x}$ is allowable or not, we form a "forbidden" matrix $F[1..n, 1..\sigma]$ in which $F[i, \lambda] = 1$ if and only if $\lambda \in \boldsymbol{x}[i]$ is forbidden. $F$ is updated and used as follows:

- for each new letter $\lambda_{\max}$ introduced in Step (S5), $F[i, \lambda_{\max}]$ is initialized to zero for all $i \in 1..n$;
- whenever an assignment $\boldsymbol{x}[i] \xleftarrow{+} \lambda$ is made in Steps (S4) & (S5), set $F[j, \lambda] \leftarrow 1$ for every $j \in N^-[i]$ (procedure `UPDATE_F`$(i, \lambda)$) .

Figures 1.9, 1.10 and 1.11 give pseudocode for Algorithm REVENG, function LEAST and function UPDATE_F, respectively.

**procedure** REVENG $(\mathcal{P}, \boldsymbol{x}, n)$
$\lambda_{\max} \leftarrow 0$; $\boldsymbol{x} \leftarrow \emptyset^n$; $F[1..n, 1..\sigma] \leftarrow 0^{n\sigma}$
**while** $\mathtt{top}(L^+) \neq \emptyset$ **do**
  $(i, j) \leftarrow \mathtt{pop}(L^+)$; $C \leftarrow \emptyset$ ▷ $i < j$; $ni+j$ *a minimum*
  **if** $\boldsymbol{x}[i] \cap \boldsymbol{x}[j] = \emptyset$ **then**
    $\forall \lambda \in \boldsymbol{x}[i]$ **do** $C_1 \overset{+}{\leftarrow} (\lambda, j)$ ▷ *ordered by* $\lambda$
    $\forall \lambda \in \boldsymbol{x}[j]$ **do** $C_2 \overset{+}{\leftarrow} (\lambda, i)$ ▷ *ordered by* $\lambda$
  ▷ *Merge* $C_1$ *and* $C_2$ *into a single sequence ordered by* $\lambda$.
    $C \leftarrow \mathtt{MERGE}(\mathtt{C_1}, \mathtt{C_2})$
    $SET \leftarrow \mathtt{false}$
    **while** $\mathtt{top}(C) \neq \emptyset$ **and not** $SET$ **do**
      $(\lambda, h) \leftarrow \mathtt{pop}(C)$
      **if** $F[h, \lambda] \neq 1$ **then**
        $\boldsymbol{x}[h] \overset{+}{\leftarrow} \lambda$ ▷ *maintain* $\lambda$ *ordering*
        $SET \leftarrow \mathtt{true}$; $\mathtt{UPDATE\_F}(h, \lambda)$
    **if not** $SET$ **then**
      $\lambda_{\max} \leftarrow \lambda_{\max}+1$
      **for** $h \leftarrow 1$ **to** $n$ **do** $F[h, \lambda_{\max}] \leftarrow 0$
      $\boldsymbol{x}[i] \overset{+}{\leftarrow} \lambda_{\max}$; $\mathtt{UPDATE\_F}(i, \lambda_{\max})$
      $\boldsymbol{x}[j] \overset{+}{\leftarrow} \lambda_{\max}$; $\mathtt{UPDATE\_F}(j, \lambda_{\max})$
**for** $i \leftarrow 1$ **to** $n$ **do**
  **if** $\boldsymbol{x}[i] = \emptyset$ **then**
  ▷ *Identify the least letter* $\lambda$ *that does* **not** *occur*
  ▷ *in* **any** $\boldsymbol{x}[j]$ *for which* $j \in N^-[i]$.
    $\lambda \leftarrow \mathtt{LEAST}(\mathtt{i}, \lambda_{\max})$; $\lambda_{\max} \leftarrow \max(\lambda, \lambda_{\max})$
    $\boldsymbol{x}[i] \leftarrow \lambda$

Figure 1.9: Given the preprocessing outlined in (DS1)-(DS2), Algorithm REVENG computes $\boldsymbol{x}[1..n]$, the lexicographically least string corresponding to a given prefix (feasible) graph $\mathcal{P}$ on $n$ vertices.

**function** LEAST(i, $\lambda_{\max}$)
$B[1..\lambda_{\max}] \leftarrow 0^{\lambda_{\max}}$
$\forall j \in N^-[i]$ **do**
    $\forall \lambda \in \boldsymbol{x}[j]$ **do**
        $B[\lambda] \leftarrow 1$
$\lambda \leftarrow 1$
**while** $\lambda \leq \lambda_{\max}$ **and** $B[\lambda] = 1$ **do**
    $\lambda \leftarrow \lambda{+}1$

Figure 1.10: Identify the least letter $\lambda$ that does **not** occur in **any** $\boldsymbol{x}[j]$ for which $j \in N^-[i]$.

**function** UPDATE_F(F, i, $\lambda$)
$\forall j \in N^-[i]$ **do**
    $F[j, \lambda] \leftarrow 1$

Figure 1.11: Update the "forbidden" matrix $F[1..n, 1..\sigma]$, whenever an assignment $\boldsymbol{x}[i] \overset{+}{\leftarrow} \lambda$ is made, set $F[j, \lambda] \leftarrow 1$ for every $j \in N^-[i]$.

### 1.3.2 Correctness

Consider first the main **while** loop of Algorithm REVENG, in which the edges of $E^+$ are considered in strict increasing $(i, j)$ order. We see that new letters $\lambda_{\max}$ are first introduced at the leftmost possible positions in $\boldsymbol{x}$. Thereafter, whenever a letter is reused ($\lambda_{\max}$ not increased), it is always the minimum possible letter consistent with the least possible currently unfilled positions $(i, j)$ that is used — by virtue of the fact that the entries in $C$ are maintained in increasing order of $\lambda$. Thus any automorphism of the alphabet $\Sigma$ other than the identity would yield a larger string. We conclude that within the main **while** loop the assignments maintain lexicographical order $\prec$ as defined in Section 1.2.

It may happen, however, that certain positions $i$ in $\boldsymbol{x}$ remain empty, those corresponding to isolated vertices $i$ in $\mathcal{P}^+$. Assignments to these positions are handled by the final **for** loop, which we now consider.

**Lemma 1.3.1** *A vertex $i \in 1..n$ is isolated in $\mathcal{P}^+$ if and only if*

*(a)* $\boldsymbol{y}[i] = 0$ *or* $i = 1$*;* **and**
*(b) for every* $j \in 2..n$, $\boldsymbol{y}[j] < i$*;* **and**
*(c) for every* $j \in 1..i{-}1$, $j{+}\boldsymbol{y}[j] \leq i$.

*Proof.* First suppose that $i$ is isolated. Then (a) must hold; otherwise, for $i > 1$ and $\boldsymbol{y}[i] > 0$, there exists an edge $(1, i) \in E^+$, a contradiction. If (b) does not hold, there exists $j > 1$ such that $\boldsymbol{y}[j] = r \geq i$, implying $\boldsymbol{x}[1..r] \approx \boldsymbol{x}[j..j+r-1]$, hence $\boldsymbol{x}[i] \approx \boldsymbol{x}[j+i-1]$, so that $(i, j+i-1) \in E^+$, again a contradiction. Similarly, if (c) does not hold, there exists $j \in 1..i-1$ such that $\boldsymbol{y}[j] = r$ with $j+r > i$. Consequently, $\boldsymbol{x}[j..j+r-1] \approx \boldsymbol{x}[1..r]$ implying $\boldsymbol{x}[i] \approx \boldsymbol{x}[i-j+1]$, so that $(i-j+1, i) \in E^+$, a contradiction that establishes sufficiency.

Suppose then that conditions (a)-(c) all hold. If we assume that $i = 1$ in (a), then (b) implies that for every $j \in 2..n$, $\boldsymbol{y}[j] = 0$, so that $E^+ = \emptyset$ and so every position $i$ is isolated. Otherwise, if $\boldsymbol{y}[i] = 0$ for some $i > 1$, conditions (b) and (c) ensure that position $i$ is not contained in any matching range within $\boldsymbol{x}$ and is therefore isolated in $\mathcal{P}^+$, as required. □

Since in the main **while** loop new maximum letters $\lambda_{\max}$ are introduced into pairs $i$ and $j > i$ of positions in $\boldsymbol{x}$ that are determined by entries in $\boldsymbol{y}$, it is an immediate consequence of Lemma 1.3.1 that $i$ must be less than any isolated vertex, in particular the smallest one, $i_{\min}$, say. In other words, every letter assigned during the execution of the main **while** loop occurs at least once to the left of $i_{\min}$ in $\boldsymbol{x}$. It follows that lexicographical order will be maintained if any required additional letters $\lambda_{\max}+1, \lambda_{\max}+2, \cdots$ are assigned to an ascending sequence of positions in $\boldsymbol{x}$ determined by the isolated vertices in $\mathcal{P}^+$. We have

**Lemma 1.3.2** *Given a prefix graph $\mathcal{P}_{\boldsymbol{y}}$ corresponding to a feasible array $\boldsymbol{y}$, Algorithm* REVENG *constructs a lexicographically least indeterminate string on a minimum alphabet whose prefix table $\boldsymbol{\pi} = \boldsymbol{y}$.*

### 1.3.3 Asymptotic Complexity

The main **while** loop in Algorithm REVENG will be executed exactly $|E^+|$ times. Within the loop the construction of $C$ requires time proportional to $|C| = |\boldsymbol{x}[i]|+|\boldsymbol{x}[j]|$, thus $O(\sigma)$ in the worst case. The processing of $C$ then also requires $O(\sigma)$ time in the worst case, except for the time required for `UPDATE_F`. Each of the three calls of `UPDATE_F` corresponds to the assignment of a letter $\lambda$ to a vertex $i$ of $\mathcal{P}^-$ and the

ensuing update of $F[i, \lambda]$, an event that can occur at most $\sigma$ times for each edge in $E^-$, thus at most $(\sigma \times |E^-|)$ times overall. Similarly, the **for** loop that initializes the $F$ array requires $\Theta(n)$ time for each of at most $\sigma$ values of $\lambda_{\max}$.

We conclude that the worst-case time requirement for the **while** loop (Figure 1.9) is $O\big(\sigma \max(|E^+|, |E^-|)\big)$. As illustrated by the examples $\boldsymbol{y} = n0^{n-1}$ and $\boldsymbol{y} = n|n-1|\ldots|1$, the bounds on these quantities are as follows: $0 \le |E^+| \le \binom{n}{2}$ and $0 \le |E^-| \le n-1$, while $|E^+|+|E^-| \ge n-1$. As noted earlier, it was shown in [BSBW14] that $\sigma \le n+\sqrt{n}$, with a further conjecture that $\sigma \le n$.

Turning our attention to the terminating **for** loop of Algorithm REVENG, we observe that for at most $n$ executions of function `LEAST`, the binary array $B$ of length at most $\sigma$ must be created, thus overall consuming $O(\sigma n)$ time. The nested $\forall$ loops in `LEAST` set positions in $B$ $\lambda$ times for at most every edge in $E^-$, again requiring at most $O(\sigma n)$ time over all invocations of `LEAST`. Thus

**Lemma 1.3.3** *Algorithm* REVENG *requires $O(\sigma n^2)$ time in the worst case, where $\sigma \le n+\sqrt{n}$.*

### 1.3.4 Example

Suppose $\boldsymbol{y} = 50210$, so that $E^+ = 13, 14, 24$ and $E^- = 12, 15, 25, 35$.

- In $E^+$ first consider edge 13, leading to assignments $\boldsymbol{x}[1] \leftarrow a$, $\boldsymbol{x}[3] \leftarrow a$, with $F[2, a] = F[5, a] = 1$ since 12, 15 and 35 are edges of $E^-$.
- Edge 14 of $E^+$ leads to $\boldsymbol{x}[4] \leftarrow a$ and no new values in $F$, since vertex 4 is isolated in $E^-$.
- Edge 24 of $E^+$ requires a new letter because $F[2, a] = 1$. Therefore we assign $\boldsymbol{x}[2] \leftarrow b$ and $\boldsymbol{x}[4] \xleftarrow{+} b$, while setting $F[1, b] = F[5, b] = 1$ because of the edges 21 and 25 in $E^-$.
- Finally we deal with the isolated vertex 5 in $E^+$ by setting $\boldsymbol{x}[5] \leftarrow c$ since $15 \in E^-$ and $\boldsymbol{x}[1] = a$, while $25 \in E^-$ and $\boldsymbol{x}[2] = b$.

The lexicographically least string is $\boldsymbol{x} = aba\{a, b\}c$.

### 1.3.5 Computational Experiments

To get an idea of how the algorithm behaves in practice, we have implemented Algorithm REVENG and conducted a simple experimental study. A set of 1,000 feasible arrays having lengths $10, 20, .., 100$ has been randomly generated as follows. For each feasible array $\boldsymbol{y}$ we randomly select a value for $\boldsymbol{y}[i], i \in [1..n]$ from within the range $[0..n - i + 1]$. The experiments have been run on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. We have implemented Algorithm REVENG in $C\#$ language using Visual Studio 2010. As is evident from Figure 1.12, the experiments suggest that average case time also increases by a factor somewhat greater than $n^2$.

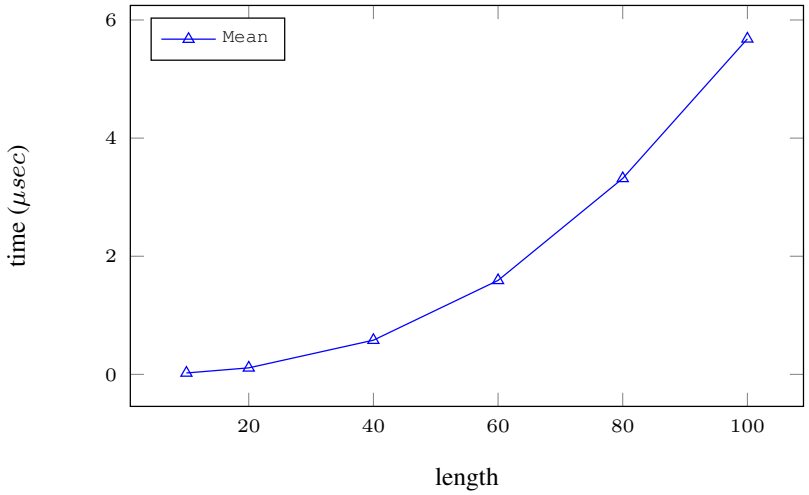



Figure 1.12: Timing results for randomly-generated feasible arrays $\boldsymbol{y}$.

## 1.4 Discussion

The high worst-case time complexity of the algorithm described here suggests room for improvement. On the other hand, it is difficult to imagine an algorithm that could do the same computation without considering all the edges of $E^+$ and thus necessitating $\Theta(n^2)$ time for many instances of the prefix table $\boldsymbol{\pi}$. Similarly, the requirement to achieve a lexicographically least solution leads to a recurring dependence on alphabet

size $\sigma$ that expresses itself in the time complexity. Even though it may be true that $\sigma \leq n$, nevertheless it seems clear that $\sigma$ can be much larger than in the regular case, where it has been shown [CCR09, CRSW14] that $\sigma \leq \lceil \log_2 n \rceil$.

We have tried approaches that focus on $E^-$ rather than $E^+$ as the primary data structure, but without success. In particular, we have considered "triangles" $i_1 j i_2$, where both $(i_1, j)$ and $(i_2, j)$ are edges in $E^+$, while $(i_1, i_2) \in E^-$, a situation that forces a string to be indeterminate. It turns out, however, that the number of such triangles is $O(n^2)$. Similarly, the ingenious graph proposed in [BSBW14], whose chromatic number is the minimum alphabet size $\sigma$ of a string corresponding to a given prefix table, has $O(n^2)$ vertices in the worst case.

At the same time, we have no proof that our algorithm is asymptotically optimal; for example, an algorithm that could eliminate the $\sigma$ factor in the complexity would be of considerable interest. Also interesting would be an algorithm for indeterminate strings that would execute in $\Theta(n)$ time on regular strings as a special case, thus matching the algorithm of [CCR09]. More generally, we propose the study of indeterminate strings ("strings" as we have called them here), their associated data structures (such as the prefix table), and their applications as a promising research area in both combinatorics on words and string algorithms.

# Article: # 2

# Computing Covers Using Prefix Tables

An ***indeterminate string*** $\boldsymbol{x} = \boldsymbol{x}[1..n]$ on an alphabet $\Sigma$ is a sequence of nonempty subsets of $\Sigma$; $\boldsymbol{x}$ is said to be ***regular*** if every subset is of size one. A proper substring $\boldsymbol{u}$ (a substring of $\boldsymbol{x}$ is a proper substring if it is not the empty word or $\boldsymbol{x}$ itself) of regular $\boldsymbol{x}$ is said to be a ***cover*** of $\boldsymbol{x}$ iff for every $i \in 1..n$, an occurrence of $\boldsymbol{u}$ in $\boldsymbol{x}$ includes $\boldsymbol{x}[i]$. The ***cover array*** $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ of $\boldsymbol{x}$ is an integer array such that $\boldsymbol{\gamma}[i]$ is the longest cover of $\boldsymbol{x}[1..i]$. Fifteen years ago a complex, though nevertheless linear-time, algorithm was proposed to compute the cover array of regular $\boldsymbol{x}$ based on prior computation of the border array of $\boldsymbol{x}$. In this article, we first describe a linear-time algorithm to compute the cover array of regular $\boldsymbol{x}$ based on the prefix table of $\boldsymbol{x}$. We then extend this result to indeterminate strings.

---

## 2.1 Introduction

The idea of a ***quasiperiod*** or ***cover*** of a string $\boldsymbol{x}$ was introduced almost a quarter-century ago by Apostolico & Ehrenfeucht [AE90]: a proper substring $\boldsymbol{u}$ of $\boldsymbol{x}$ such that every position in $\boldsymbol{x}$ lies within an occurrence of $\boldsymbol{u}$. Thus, for example, $\boldsymbol{u} = aba$ is a cover of $\boldsymbol{x} = ababaababa$. In [AFI91] a linear-time algorithm was described to compute the shortest cover of $\boldsymbol{x}$; this contribution was followed by linear-time algorithms to compute

- the shortest cover of every prefix of $\boldsymbol{x}$ [Bre92];
- all the covers of $\boldsymbol{x}$ [MS94, MS95];
- all the covers of every prefix of $\boldsymbol{x}$ [LS02].

A ***border*** of a string $\boldsymbol{x}$ is a possibly empty proper prefix of $\boldsymbol{x}$ that is also a suffix of $\boldsymbol{x}$. (Thus a cover of $\boldsymbol{x}$ is necessarily also a border of $\boldsymbol{x}$.) In the ***border array*** $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ of the string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, $\boldsymbol{\beta}[i]$ is the length of the longest border of $\boldsymbol{x}[1..i]$. Since for $\boldsymbol{\beta}[i] \neq 0$, $\boldsymbol{\beta}[\boldsymbol{\beta}[i]]$ is the length of a border of $\boldsymbol{x}$ as well as the length of the longest border of $\boldsymbol{x}[1..\boldsymbol{\beta}[i]]$ [AHU74, Smy03], it follows that $\boldsymbol{\beta}$ provides all the borders of every prefix of $\boldsymbol{x}$. For example:

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{x} = & a & b & a & b & a & b & a & a & b & a \\ \boldsymbol{\beta} = & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 1 & 2 & 3 \end{array} \tag{2.1}$$

As shown in [LS02], the ***cover array*** $\boldsymbol{\gamma}$ has a similar cascading property, giving the lengths of all the covers of every prefix of $\boldsymbol{x}$ in a compact form:

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{\gamma} = & 0 & 0 & 0 & 2 & 3 & 4 & 5 & 0 & 0 & 3 \end{array}$$

Here $\boldsymbol{x}[1..7]$ has covers $\boldsymbol{u_1} = \boldsymbol{x}[1..5] = ababa$ and $\boldsymbol{u_2} = \boldsymbol{x}[1..3] = aba$, while the entire string $\boldsymbol{x}$ has cover $\boldsymbol{u_2}$. The main result of [LS02] is an algorithm that computes $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ from $\boldsymbol{\beta} = \boldsymbol{\beta}[1..n]$ in $\Theta(n)$ time, while making no reference to the underlying string $\boldsymbol{x}$.

The results outlined above all apply to a ***regular string*** — that is, a string $\boldsymbol{x}$ such that each entry $\boldsymbol{x}[i]$ is constrained to be a one-element subset of a given set $\Sigma$ called the ***alphabet***. In this article, we show how to extend these ideas and algorithms to an ***indeterminate string*** $\boldsymbol{x}$ — that is, such that each $\boldsymbol{x}[i]$ can be any nonempty subset of $\Sigma$. Observe that every regular string is indeterminate.

The idea of an indeterminate string was first introduced in [FP74], then studied further in the 1980s as a "generalized string" [Abr87]. Over the last 15 years Blanchet-Sadri has written numerous papers on the properties of "strings with holes" (each $\boldsymbol{x}[i]$ is either a one-element subset of $\Sigma$ or $\Sigma$ itself), together with a monograph on the subject [BS08]; while other authors have studied indeterminate strings in their full generality, together with related algorithms [BRS09, NRR12, HS03, HSW08, SW08, SW09b, SW09a, CRSW14]. In the specific context of this article, Voráček & Melichar [VM05] have done pioneering work on the computation of covers and related structures in generalized strings using finite automata.

For indeterminate strings, equality of letters is replaced by the idea of a "match" [HS03]: $\boldsymbol{x}[i]$ ***matches*** $\boldsymbol{x}[j]$ (written $\boldsymbol{x}[i] \approx \boldsymbol{x}[j]$) if and only if $\boldsymbol{x}[i] \cap \boldsymbol{x}[j] \neq \emptyset$, while $\boldsymbol{x} \approx \boldsymbol{y}$ if and only if $|\boldsymbol{x}| = |\boldsymbol{y}|$ and corresponding positions in $\boldsymbol{x}$ and $\boldsymbol{y}$ all match. It is important to note that matching is nontransitive: $b \approx \{b, c\} \approx c$, but $b \not\approx c$.

It is [CRSW14] that provides the point of departure for our contribution, as we now explain. The ***prefix table*** $\boldsymbol{\pi} = \boldsymbol{\pi}[1..n]$ of $\boldsymbol{x}[1..n]$ is an integer array such that $\boldsymbol{\pi}[1] = n$ and, for every $i \in 2..n$, $\boldsymbol{\pi}[i]$ is the length of the longest substring occurring at position $i$ of $\boldsymbol{x}$ that matches a prefix of $\boldsymbol{x}$. Thus, for our example (2.1):

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $a$ | $b$ | $a$ |
| $\boldsymbol{\pi} =$ | 10 | 0 | 3 | 0 | 3 | 0 | 1 | 3 | 0 | 1 |

It turns out [BKS13] that the prefix table and the border array are "equivalent" for regular strings; that is, each can be computed from $\boldsymbol{x}$ in linear time, and each can be computed from the other, without reference to $\boldsymbol{x}$, also in linear time. However, for indeterminate strings, this is not true: the prefix table continues to determine all the borders of every prefix of $\boldsymbol{x}$, while the border array, due to the intransitivity of matching, is no longer reliable in identifying borders shorter than the longest one. Consider, for example:

$$\begin{array}{lccc} & 1 & 2 & 3 \\ \boldsymbol{x} = & a & \{a, b\} & b \\ \boldsymbol{\beta} = & 0 & 1 & 2 \end{array}$$

Here $\boldsymbol{x}$ does not have a border of length $\boldsymbol{\beta}[\boldsymbol{\beta}[3]] = 1$; on the other hand, $\boldsymbol{\pi} = 320$ correctly identifies all the borders of every prefix of $\boldsymbol{x}$.

Moreover, it was shown in [CRSW14] that every ***feasible*** array — that is, every array $\boldsymbol{y} = \boldsymbol{y}[1..n]$ such that $\boldsymbol{y}[1] = n$ and for every $i \in 2..n$, $\boldsymbol{y}[i] \in 0..n{-}i{+}1$ — is a prefix table of some (indeterminate) string. Thus there exists a many-many correspondence between all possible prefix tables and all possible indeterminate strings. Furthermore, [SW08] describes an algorithm to compute the prefix table of any indeterminate string, while [ARS15] gives an algorithm to compute a lexicographically least indeterminate string corresponding to a given prefix table.

At this point let us discuss our motivation more precisely. First, realize that to exploit the fullest functionality of a border array of an indeterminate string we need to resort to the extended definition of the border array which in fact requires quadratic space [HS03, NRR12, BRS09]: unlike the border array of a regular string, which is a simple array of integers, the border array of an indeterminate string is an array of lists of integers. Here at each position, the list gives all possible borders for that prefix. On the other hand, the prefix array, even for the indeterminate string, remains a simple one-dimensional array, just as for a regular string. It thus becomes of interest to make use of the prefix table rather than the border array whenever possible, in order to extend the scope of computations to indeterminate strings.

In Section 2.2 of this article, we describe a linear-time algorithm to compute the cover array $\boldsymbol{\gamma}$ of a regular string $\boldsymbol{x}$ directly from its prefix table $\boldsymbol{\pi}$. Then, Section 2.3 describes a limited extension of this algorithm to indeterminate strings. Finally, Section 1.4 outlines future research directions, especially making use of prefix tables to extend the utility and applicability of other data structures to indeterminate strings.

## 2.2 Prefix-to-Cover for a Regular String

In this section we describe our basic $\Theta(n)$-time algorithm `PCR` to compute the cover array $\boldsymbol{\gamma} = \boldsymbol{\gamma}[1..n]$ of a regular string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ directly from its prefix table $\boldsymbol{\pi} =$

$\boldsymbol{\pi}[1..n]$. In fact, as noted in the Introduction, $\boldsymbol{\gamma}$ actually provides all the covers of every prefix of $\boldsymbol{x}$. Central to our algorithm are the following definitions:

**Definition 21** *If, for a position $i \in 1..n$, $\boldsymbol{\pi}[i] > 0$, then $R_i = [i, i+\boldsymbol{\pi}[i]-1]$ is said to be the **range** at $i$ of **length** $\boldsymbol{\pi}[i]$; the ranges $R_i$ and $R_{i'}$, $i' > i$, are **connected** if and only if $i' \leq i+\boldsymbol{\pi}[i] < i'+\boldsymbol{\pi}[i']$.*

Notably, in what follows, for the sake of brevity, we may slightly abuse the notation $R_i = [i, i+\boldsymbol{\pi}[i]-1]$ by simply saying $R_i = \boldsymbol{\pi}[i]$.

**Definition 22** *Position $j$ in $\boldsymbol{\pi}$ is said to be **live** at position $i' > j$ if and only if there exists a sequence of $h \geq 1$ connected ranges $R_{i_1}, R_{i_2}, \ldots, R_{i_h}$, each of length at least $j$, such that $i_1 \leq j+1$, $i_h+\boldsymbol{\pi}[i_h]-1 \geq i'$. Otherwise, $j$ is said to be **dead** at $i'$.*

Thus $\boldsymbol{x}[1..n]$ has a cover $\boldsymbol{x}[1..j]$, $j < n$, if and only if $j$ is live at $n$ and the final connected range $R_{i_h}$ satisfies $i_h+\boldsymbol{\pi}[i_h]-1 = n$.

For a positive integer $n$, let $[1, n] = \{1, . . , n\}$. A set $\{[b_1, e_1], . . , [b_h, e_h]\}$ of sub-intervals of $[1, n]$ is called a cover of interval $[1, n]$, if $\bigcup_{i=1}^{h}[b_i, e_i] = [1, n]$. The size of the cover is the number $h$ of sub-intervals in it.

The strategy of Algorithm PCR (Figure 2.1) is to perform an on-line left-to-right scan of $\boldsymbol{\pi}$, identifying connected ranges $R_i$. This process may be complex. Within range $R_i$ there may exist two (or more) positions $i_1 > i$ and $i_2 > i_1$ that define ranges $R_{i_1}$ and $R_{i_2}$, both connected to $R_i$; of these, PCR processes $R_i$ first, followed by $R_{i_1}$, then, if $R_{i_1}$ and $R_{i_2}$ are connected (they may not be), by $R_{i_2}$. For example, consider

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ |
| $\boldsymbol{\pi} =$ | 19 | 0 | 5 | 0 | 3 | 0 | 1 | 7 | 0 | 7 | 0 | 7 | 0 | 6 | 0 | 4 | 0 | 2 | 0 |
| $\boldsymbol{\gamma} =$ | 0 | 0 | 0 | 2 | 3 | 4 | 5 | 0 | 0 | 3 | 0 | 5 | 0 | 7 | 0 | 7 | 0 | 7 | 0 |

(2.2)

Here the pairs of ranges $(R_8, R_{10})$, $(R_8, R_{12})$ and $(R_{10}, R_{12})$ are all connected: PCR will process positions 8–14 in $R_8$, followed by 15–16 in $R_{10}$, then 17–18 in $R_{12}$ and finally position 19 in $R_{14}$.

Algorithm PCR processes each connected range $R_i$ twice, first in left-to-right order, beginning at position $i' = lastlim+1$, where $lastlim$ is the current rightmost position

```
procedure PCR (π, γ)
γ[1..n] ← 0^n; maxlive[1..n] ← 0^n
lastlim ← 1; i ← 2
while lastlim < n do
    j ← π[i]
    if j = 0 then
    ▷ No range extends beyond lastlim, so 1, 2, ..., i−1 are all dead.
        if i > lastlim then
            maxlive[i−1] ← −1; lastlim ← i
    else
        lim ← i+j−1
        if lim > lastlim then
            j′ ← (lastlim+1) − i
        ▷ Initial setting of maxlive and γ.
        for i′ ← lastlim+1 to lim do
            j′ ← j′+1
            if (maxlive[j′] = 0 and i′ ≤ 2j′)
            or maxlive[j′] ≥ i′−j′ then
            ▷ j′ is a cover of x[1..i′].
                maxlive[j′] ← i′; γ[i′] ← j′
            else
            ▷ j′ is ruled out as a cover.
                maxlive[j′] ← −1
        ▷ Reset maxlive and γ in case of multiple covers.
        for i′ ← lim downto lastlim+1 do
            j″ ← γ[j′]
            ▷ A cover of x[1..j′] is also a cover of x[1..i′].
            while j″ > 0 and 0 < maxlive[j″] < i′ do
                maxlive[j″] ← i′; γ[i′] ← max(γ[i′], j″)
                j″ ← γ[j″]
            j′ ← j′−1
        lastlim ← lim
    i ← i+1
```

Figure 2.1: Compute the cover array $\gamma$ of a regular string $x$ from its prefix table $\pi$.

for which $\boldsymbol{\gamma}$ has already been determined, and ending at $i' = lim > lastlim$, the rightmost position in $R_i$. Corresponding to each $i'$ is the length $j' = i' - i + 1$ of the prefix of $R_i$ (hence also of $\boldsymbol{x}$) that may extend a sequence of covering substrings of length $j'$. In order to determine whether or not $j'$ is live at $i'$, PCR maintains an array $maxlive[1..n]$, using the following values:

$$\begin{aligned} maxlive[j'] = 0 &\ : \text{ initial setting: position } j' \text{ not yet considered} \\ i' &\ : \ j' \text{ live at } i' \text{: } \boldsymbol{x}[1..i'] \text{ covered by } \boldsymbol{x}[1..j'] \\ -1 &\ : \ j' \text{ is (permanently) dead} \end{aligned}$$

However, it can happen that $maxlive$ and $\boldsymbol{\gamma}$ are not correctly set by the left-to-right scan of $R_i$:

We will use the notion $\boldsymbol{\gamma}^1[i] = \boldsymbol{\gamma}[i]$, $1 \le i \le n$ with $\boldsymbol{\gamma}^j[i] = \boldsymbol{\gamma}[\boldsymbol{\gamma}^{j-1}[i]]$ for every $j \ge 2$ such that $\boldsymbol{\gamma}^{j-1}[i]$ is defined and $1 \le \boldsymbol{\gamma}^{j-1}[i] \le n$.

**Definition 23 ([LS02])** *In the cover array $\boldsymbol{\gamma}$, if there exists an integer $k \ge 1$ and positions $i > j > 0$ such that $\boldsymbol{\gamma}^k[i] = j$, then $j$ is said to be the $k^{\textbf{th}}$ **ancestor** of $i$ in $\boldsymbol{\gamma}$. Thus the cover array determines a **cover tree**.*

It may be that $\boldsymbol{\gamma}[i']$ is set to zero because $j'$ is dead at $i'$, even though an ancestor of $j'$ in the cover tree is live at $i'$; on the other hand, when $\boldsymbol{\gamma}[i'] = j'$, so that ancestors of $j'$ may also be live at $i'$, the $maxlive$ values of the ancestors may need to be adjusted. Thus a second right-to-left scan of $R_i$ is required, in order to ensure that these updates are correct.

For example, in (2.2), we need to ensure that $maxlive[5] = maxlive[3] = 18$, since both 5 and 3 are live ancestors of 7. A more subtle example is given in (2.3), where at position 19 we need to recognize that both 5 and 3 are live, even though 7 is dead, so that later, at position 22, we can recognize that 3 is live:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ |
| $\boldsymbol{\pi} =$ | 22 | 0 | 5 | 0 | 3 | 0 | 1 | 5 | 0 | 3 | 0 | 1 | 7 | 0 | 5 | 0 | 3 | 0 | 1 | 3 | 0 | 1 |
| $\boldsymbol{\gamma} =$ | 0 | 0 | 0 | 2 | 3 | 4 | 5 | 0 | 0 | 3 | 0 | 5 | 0 | 0 | 3 | 0 | 5 | 0 | 5 | 0 | 0 | 3 |

(2.3)

Consider also

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x} =$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ | $a$ | $b$ | $b$ | $a$ | $b$ | $a$ | $b$ | $a$ | $b$ |
| $\boldsymbol{\pi} =$ | 22 | 0 | 5 | 0 | 3 | 0 | 1 | 7 | 0 | 7 | 0 | 5 | 0 | 3 | 0 | 1 | 5 | 0 | 3 | 0 | 1 | 7 | 0 | 5 | 0 | 3 | 0 | 1 |
| $\boldsymbol{\gamma} =$ | 0 | 0 | 0 | 2 | 3 | 4 | 5 | 0 | 0 | 3 | 0 | 5 | 0 | 7 | 0 | 7 | 0 | 0 | 3 | 0 | 5 | 0 | 0 | 3 | 0 | 5 | 0 | 5 |

(2.4)

Thus, using $n$ additional words of storage and a double scan of each connected range, Algorithm PCR is able to compute $\boldsymbol{\gamma}$. The time requirement is $\Theta(2n)$ plus the time required by the internal **while** loop; this loop updates $maxlive[j']$ at most once for each ancestral position $j'$ in the range, thus requiring a total $\mathcal{O}(n)$ time overall. Hence we have the following result:

**Theorem 2.2.1** *Given the prefix table $\boldsymbol{\pi}$ of a regular string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, Algorithm* PCR *correctly computes the cover array $\boldsymbol{\gamma}$ of $\boldsymbol{x}$ in $\Theta(n)$ time using an additional $n$ integers of space.*

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{s}$ | a | b | a | a | b | a | b | a | a | b | a | a | b | a | b | a | a | b | a | b | a | b | a |
| $\pi$ | 23 | 0 | 1 | 3 | 0 | 6 | 0 | 1 | 11 | 0 | 1 | 3 | 0 | 8 | 0 | 1 | 3 | 0 | 3 | 0 | 3 | 0 | 1 |
| $\gamma$ | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 3 | 0 | 5 | 6 | 0 | 5 | 6 | 0 | 8 | 9 | 10 | 11 | 0 | 8 | 0 | 3 |

Figure 2.2: The prefix and cover array of $\boldsymbol{s} = abaababaabaababaababa$.

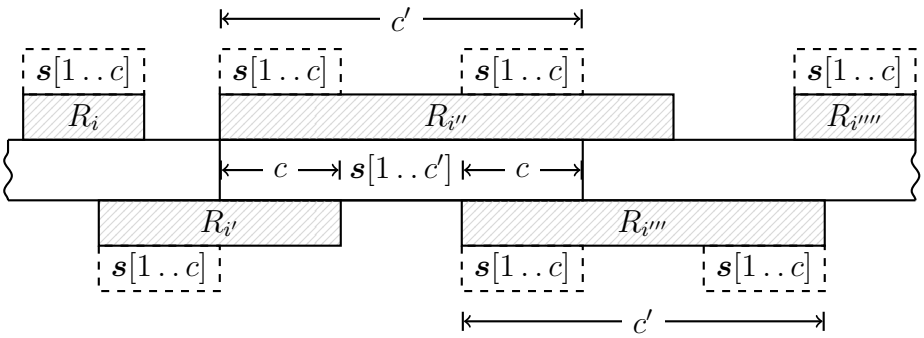



Figure 2.3: Showing two covers from $\gamma(\boldsymbol{x})$, $\boldsymbol{x} = abaababaabaababaababa$ (2.2)

## 2.3 Extensions to Indeterminate Strings

It turns out that for indeterminate strings there are two natural analogues of the idea of "cover".

**Definition 24** *A string $\boldsymbol{x} = \boldsymbol{x}[1..n]$ is said to have a **sliding cover** of length $\kappa$ if and only if*

*(a) $\boldsymbol{x}$ has a suffix $\boldsymbol{v}$ of length $|\boldsymbol{v}| = \kappa$; and*

*(b) $\boldsymbol{x}$ has a proper prefix $\boldsymbol{u}$, $|\boldsymbol{u}| \geq |\boldsymbol{x}| - \kappa$, with suffix $\boldsymbol{v'} \approx \boldsymbol{v}$; and*

*(c) either $\boldsymbol{u} = \boldsymbol{v'}$ or else $\boldsymbol{u}$ has a cover of length $\kappa$.*

A sliding cover requires that adjacent or overlapping substrings of $\boldsymbol{x}$ match, but the nontransitivity of matching leaves open the possibility that nonadjacent elements of the cover do not match. For example,

$$\boldsymbol{x} = \{a, b\}c\{a, c\}\{a, c\}ca \tag{2.5}$$

has a sliding cover of length $\kappa = 2$ because $\{a, b\}c \approx \{a, c\}\{a, c\} \approx ca$, even though $\{a, b\}c \not\approx ca$.

However, note that the very concept of "regularity of a string" in some sense breaks down when we consider the concept of a sliding cover: now the "cover" need not actually "match" the area it is covering. In fact, the above concept even allows for a string to be a cover of an indeterminate string without being a substring of the latter at all! This motivates the idea of a ***rooted cover*** of length $\kappa$, where every covering substring is required to match, not the preceding entry in the cover, but rather the prefix of $\boldsymbol{x}$ of length $\kappa$. A rooted cover is defined simply by changing "suffix" to "prefix" in part (b) of Definition 24. The example string (2.5) has no rooted cover, but the string $\boldsymbol{x'} = \{a, b\}c\{a, c\}\{a, c\}ac$ has both a sliding cover and a rooted cover of length 2. Notably, in the literature, the concept of rooted cover is in fact used as the cover for an indeterminate string [BRS09].

### 2.3.1 Computing Rooted Covers

In this section we describe Algorithm PCInd (Fig. 2.4) to compute the set of rooted covers $\Gamma$ of a given indeterminate string $\boldsymbol{x} \in \Sigma^n$ directly from its prefix table. As will be shown below, the algorithm runs in linear time on average and $\mathcal{O}(n^2)$ time in the worst case.

Algorithm PCInd maintains a list $\mathcal{L}$ to store the candidate rooted covers. The algorithm also maintains an auxiliary push-down store $\mathcal{D}$, which stores the list of dead covers at each iteration $i \in [2..n]$. The push-down store $\mathcal{D}$ will be used for marking the dead covers so as to delete them at the end of each iteration. Lastly, in order to determine whether or not the cover of length $v$ is live at position $i$, the algorithm maintains an array $maxlive[1..n]$ the same as in Algorithm PCR.

Exploiting the fact that the rooted cover of an indeterminate string $\boldsymbol{x}$ is also a border of it, the algorithm starts by identifying the set of candidate (rooted) covers as defined below.

**Definition 25** *Let $\boldsymbol{x} \in \Sigma^n$ and let $\pi[1..n]$ be its prefix array. Then the set of candidate (rooted) covers $\mathcal{L}$ of the whole string $\boldsymbol{x}$ is:*

$$\mathcal{L} \subseteq \pi : \textit{ where } \pi[i] + i - 1 = n \textit{ for } 2 \leq i \leq n \tag{2.6}$$

To populate the list of candidate covers, we start by computing the value $max = \max(\pi[2..n])$. Then the algorithm initializes the list $\mathcal{L}$ with the filtered entries from the set $\{1, 2, ..., max\}$, such that $\mathcal{L}$ will only store the values that satisfies $y[i] + i - 1 = n$ for $i \in [2..n]$.

During the execution of the main **for** loop, at each position $i \in [2..n]$. The algorithm tests, for each candidate cover $v$ in list $\mathcal{L}$, whether or not $v$ is active. Based on the result of this test the algorithm appropriately updates the corresponding entry in the $maxlive$ array and marks the dead covers at position $i$, by storing those in $\mathcal{D}$ which will be deleted at the end of each iteration using a **while** loop.

After computing the array $maxlive$ (at the end of the main **for** loop), we can easily identify and report the set of rooted covers of the whole string $\boldsymbol{x}$ simply by finding all the entries in the array $maxlive$ that have the value $n$ (i.e., all entries of the list of candidate covers that are still active).

```
procedure PCInd(π, Γ)
    Γ ← φ; L ← φ; maxlive[1..n] ← 0^n
    max ← max(π[2..n])
    ▷ fill the list L with the candidate covers from {1, 2, ..., max}
    for i ← 1 to max do
        ▷ consider only border values
        if π[i] + i − 1 = |s| then
            L ←+ i
    for i ← 2 to n do
        ▷ D stores list of dead covers at position i
        D ← φ
        for all (v ∈ L) do
            ▷ skip values of v > π[i]
            if (v > π[i]) then
                break
            t ← i + v − 1
            if ((maxlive[v] = 0 and t ≤ 2 ∗ v)
                or (maxlive[v] ≥ t − v)) then
                ▷ cover v is still live
                maxlive[v] ← t
            else
                ▷ cover v is dead
                maxlive[v] ← −1
                ▷ mark cover v for deletion
                push(D) ← v
        ▷ remove the dead covers from L
        while top(D) ≠ ∅ do
            r ← pop(D)
            L ←− r
    ▷ report the rooted covers
    for i ← 1 to n do
        if maxlive[i] = n then
            Γ ←+ i
```

Figure 2.4: Compute all rooted covers of indeterminate string from its prefix array.

A final note regarding the use of the push-down store $\mathcal{D}$ is in order. The standard approach, when the programming language in use allows it, is to delete some elements from a list while iterating through it. This can be done either: (1) by iterating back-

wards through the list and then deleting within the **for** loop, or (2) by identifying all items that need to be deleted and marking them with a flag (in the first iteration), then (in the second iteration) removing all those items which are flagged for deletion. However, in both cases (1) and (2), the algorithm must loop through all the items in the list $\mathcal{L}$ after each iteration. Alternatively, keeping track of the items to remove in another list (e.g., in $\mathcal{D}$) and then, after all items have been processed, enumerating the remove list ($\mathcal{D}$) and removing each item from the list of candidate covers ($\mathcal{L}$) requires only looping through $\mathcal{D}$.

### 2.3.2 Analysis

Finding the value $max$ in $\pi[2..n]$ can be done with a simple linear scan of the array $\pi$. Computing the list $\mathcal{L}$ of candidate covers can be done in $\mathcal{O}(n)$ time. The main **for** loop will be executed exactly $n$ times.

Within the loop the checking of the condition whether a cover is active or not can be done in constant time for a particular value and hence the total testing of $live$ or $dead$ for all candidate covers requires time proportional to $|\mathcal{L}|$, which is $\mathcal{O}(n)$ in the worst case. Note that the list $\mathcal{L}$ tends to get smaller and smaller as the iteration continues, because we keep removing dead covers from it after each iteration. However, the complexity remains $\mathcal{O}(n)$ in the worst case (e.g., $\boldsymbol{x} = a^n$).

Turning our attention to the **while** loop at the end of each iteration of the main **for** loop, the processing of $\mathcal{D}$ to remove the dead covers also requires time proportional to $\mathcal{D}$, thus $\mathcal{O}(n)$ in the worst case since the total number of covers is bounded by $n$. We conclude that the worst-case time requirement for the main **for** loop is $\mathcal{O}(n^2)$. The final **for** loop to report the list of rooted covers requires time proportional to $|maxlive|$ which is $\mathcal{O}(n)$. The algorithm requires linear extra space to store the lists $maxlive$, $\mathcal{L}$ and $\mathcal{D}$. So we have the following result:

**Theorem 2.3.1** *Given the prefix table $\pi$ of an indeterminate string $\boldsymbol{x} = \boldsymbol{x}[1..n]$, Algorithm* `PCInd` *correctly computes the set of rooted covers of the whole string of $\boldsymbol{x}$ in $\mathcal{O}(n^2)$ time and linear space.*

Finally, Bari et. al. [BRS09] proved that the expected number of borders of an indeterminate string is bounded by a constant. Since, in the beginning of Algorithm

PCInd we include only the borders in $\mathcal{L}$, this means that the size of the list $\mathcal{L}$ and also $\mathcal{D}$ is bounded by a constant. Therefore, based on the analysis presented above we can conclude that Algorithm PCInd runs in linear time on average.

### 2.3.3 An Illustrative Example

Suppose $\pi = \{12, 3, 2, 1, 1, 7, 6, 1, 0, 3, 0, 1\}$. We have $max = 7$. The simulation of the algorithm is shown in Fig. 2.5. The algorithm initializes the set $\mathcal{L}$ with the set of candidate covers. Hence, we have $\mathcal{L} = \{1, 3, 6, 7\}$. At iteration $i = 6$, we can see that cover 3 becomes non-active, so the value $maxlive[3]$ is set to $-1$ and the cover 3 is removed from the set of candidate covers. Similarly, at iteration $i = 10$, the cover 1 becomes non-active, so the value $maxlive[1]$ is set to $-1$ and the cover 1 is removed from the set of candidate covers. After computing the array $maxlive$, the list of rooted covers can be identified as all the positions $i$ in $maxlive$ where $maxlive[i] = n$. So the covers are 6 and 7 since $maxlive[6] = 12$ and $maxlive[7] = 12$. We have $\Gamma = \{6, 7\}$.

| $i$ | $maxlive$ | $\mathcal{L}$ |
|---|---|---|
| 2 | $\{2, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ | $\{1, 3, 6, 7\}$ |
| 3 | $\{3, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ | $\{1, 3, 6, 7\}$ |
| 4 | $\{4, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ | $\{1, 3, 6, 7\}$ |
| 5 | $\{5, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ | $\{1, 3, 6, 7\}$ |
| 6 | $\{6, 0, -1, 0, 0, 11, 12, 0, 0, 0, 0, 0\}$ | $\{1, 6, 7\}$ |
| 7 | $\{7, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$ | $\{1, 6, 7\}$ |
| 8 | $\{8, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$ | $\{1, 6, 7\}$ |
| 9 | $\{8, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$ | $\{1, 6, 7\}$ |
| 10 | $\{-1, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$ | $\{6, 7\}$ |
| 11 | $\{-1, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$ | $\{6, 7\}$ |
| 12 | $\{-1, 0, -1, 0, 0, 12, 12, 0, 0, 0, 0, 0\}$ | $\{6, 7\}$ |

Figure 2.5: The running values of Algorithm PCInd for a given string with prefix array $\pi = \{12, 3, 2, 1, 1, 7, 6, 1, 0, 3, 0, 1\}$

### 2.3.4 The experiment

To get an idea of how the algorithm behaves in practice, we have implemented Algorithm PCInd and conducted a simple experimental study. The experiments have

been carried out on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. The algorithm have been implemented in $C\#$ language using Visual Studio 2010.

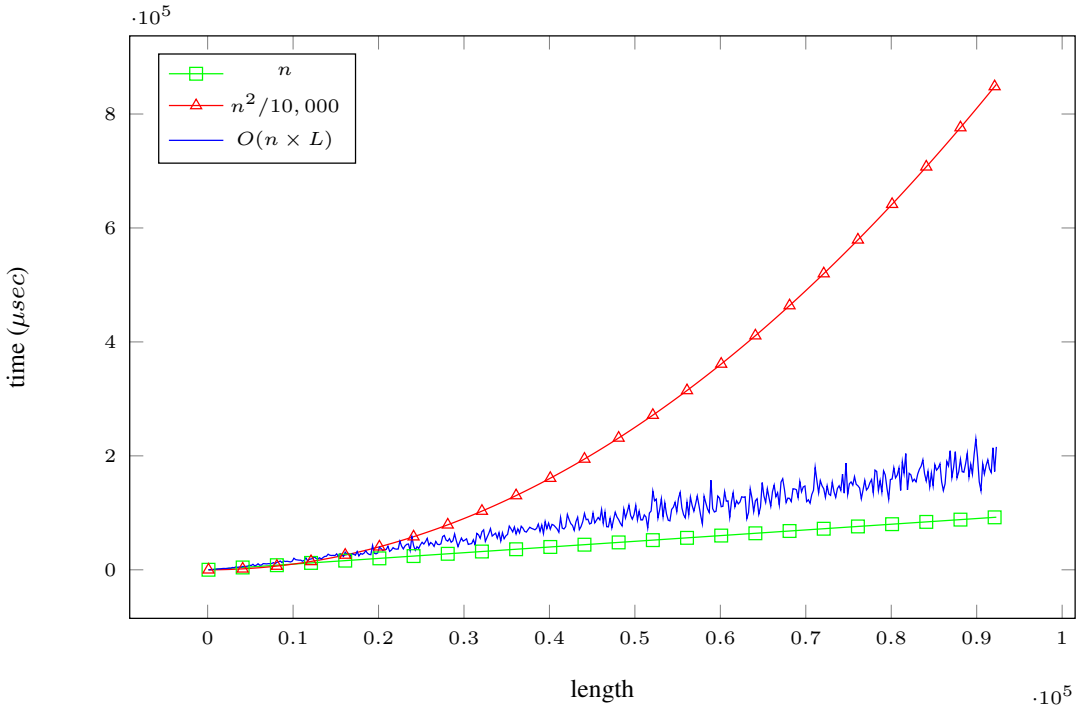



Figure 2.6: The average running time of the Algorithm `PCInd`.

We have run Algorithm `PCInd` on a set of 100 randomly generated prefix arrays for each length $n \in \{100, 200, \ldots, 100000\}$ (averaged over 100 runs for each length) and counted the average number of executions of the inner loop of the algorithm. The resulting graph (Fig. 2.6) shows the average complexity of Algorithm `PCInd` fluctuating around $n$. Note that the values $n^2$ in the graph are scaled down by $10,000$ (i.e., the curves are showing $n^2/10,000$) to have a better view of the curves. The results show that the run time of the algorithm is close to linear confirming the average case time complexity of $\mathcal{O}(n)$.

# Article: # 3

# Algorithms for Longest Common Abelian Factors

In this article, we consider the problem of computing the longest common abelian factor (LCAF) between two given strings. We present a simple $\mathcal{O}(\sigma\ n^2)$ time algorithm, where $n$ is the length of the strings and $\sigma$ is the alphabet size, and a sub-quadratic running time solution for the binary string case, both having linear space requirement. Furthermore, we present a modified algorithm applying some interesting tricks and experimentally show that the resulting algorithm runs faster.

---

## 3.1 Introduction

Abelian properties concerning words have been investigated since the very beginning of the study of Formal Languages and Combinatorics on Words. Abelian powers were first considered in 1961 by Erdős [Erd61] as a natural generalization of usual powers. In 1966, Parikh [Par66] defined a vector having length equal to the alphabet cardinality, which reports the number of occurrences of each alphabet symbol inside a given string. Later on, the scientific community started referring to such a vector as the ***Parikh vector***. Clearly, two strings having the same ***Parikh vector*** are permutations of one another and there is an ***abelian match*** between them.

Abelian properties of strings have recently grown tremendous interest among the Stringology researchers and have become an involving topic of discussion in the recent issues of the StringMasters meetings. Despite the fact that there are not so many real life applications where comparing commutative sequence of objects is relevant, abelian combinatorics has a potential role in filtering the data in order to find potential occurrences of some approximate matches. For instance, when one is looking for typing errors in a natural language, it can be useful to select the abelian matches first and then look for swap of adjacent or even near appearing letters. The swap errors and the inversion errors are also very common in the evolutionary process of the genetic code of a living organism and hence is often interesting from Bioinformatics perspective. Similar applications can also be found in the context of network communications.

In this article, we focus on the problem of finding the Longest Common Abelian Factor of two given strings. The problem is combinatorially interesting and analogous to the Longest Common Substring (LCStr) problem for the usual strings. The LCStr problem is a Historical problem and Dan Gusfield reported the following in his book [Gus97, Sec. 7.4] regarding the belief of Don Knuth about the complexity of the problem:

> [...in 1970 Don Knuth conjectured a linear time algorithm for this problem would be impossible.]

However, contrary to the above conjecture, decades later, a linear time solution for the LCStr problem was in fact obtained by using the linear construction of the suffix tree [Wei73, McC76, Ukk95]. For Stringology researchers this alone could be the

motivation for considering LCAF from both algorithmic and combinatorics point of view. However, despite a number of works on abelian matching, to the best of our knowledge, this problem has never been considered until very recently when it was posed in the latest issue of the StringMasters, i.e., StringMasters 2013. To this end, this research work can be seen as a first attempt to solve this problem with the hope of many more to follow.

A trivial brute-force solution of the LCAF problem for a fixed size alphabet has $\mathcal{O}(n^3)$ time and constant space complexity, where $n$ is the length of the longest string. In this article, we present a simple solution running in $\mathcal{O}(\sigma\ n^2)$ time, where $\sigma$ is the alphabet size. Then we present a sub-quadratic algorithm for the binary string case. Both the algorithms have linear space requirement. Furthermore, we present a modified algorithm applying some interesting tricks and experimentally show that the resulting algorithm complexity can be reduced to $\mathcal{O}(n\ \log n)$, then to $\mathcal{O}(n)$ time.

The rest of the article is organized as follows. In Sec. 3.2 we state some preliminary definitions. In Sec. 3.3 we present the quadratic time solution. In Sec. 3.5 we discuss some tricks and modify the above algorithm. We also report some experimental results that show that the modified algorithm in fact runs faster. Sec. 3.4 presents the sub-quadratic solution for the binary alphabet.

## 3.2 Preliminaries

Given a string $\boldsymbol{s}$ over the alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_\sigma\}$, we denote by $|\boldsymbol{s}|_{\alpha_j}$ the number of $\alpha_j$'s in $\boldsymbol{s}$, for $1 \leq j \leq \sigma$. We define the ***Parikh vector*** of $\boldsymbol{s}$ as $\mathcal{P}_{\boldsymbol{s}} = (|\boldsymbol{s}|_{\alpha_1}, \ldots, |\boldsymbol{s}|_{\alpha_\sigma})$.

In the binary case, we denote $\Sigma = \{0, 1\}$, the number of 0's in $\boldsymbol{s}$ by $|\boldsymbol{s}|_0$, the number of 1's in $\boldsymbol{s}$ by $|\boldsymbol{s}|_1$ and the ***Parikh vector*** of $\boldsymbol{s}$ as $\mathcal{P}_{\boldsymbol{s}} = (|\boldsymbol{s}|_0, |\boldsymbol{s}|_1)$. We now focus on binary strings. The general alphabet case will be considered later.

For a given binary string $\boldsymbol{s}$ of length $n$, we define an $n \times n$ matrix $\mathcal{M}_{\boldsymbol{s}}$ as follows. Each row of $\mathcal{M}_{\boldsymbol{s}}$ is dedicated to a particular length of factors of $\boldsymbol{s}$. So, Row $\ell$ of $\mathcal{M}_{\boldsymbol{s}}$ is dedicated to $\ell$-length factors of $\boldsymbol{s}$. Each column of $\mathcal{M}_{\boldsymbol{s}}$ is dedicated to a particular starting position of factors of $\boldsymbol{s}$. So, Column $i$ of $\mathcal{M}_{\boldsymbol{s}}$ is dedicated to the position $i$ of $\boldsymbol{s}$. Hence, $\mathcal{M}_{\boldsymbol{s}}[\ell][i]$ is dedicated to the $\ell$-length factor that starts at position $i$ of $\boldsymbol{s}$ and it reports the number of 1's of that factor. Now, $\mathcal{M}_{\boldsymbol{s}}[\ell][i] = m$ if and only if the $\ell$-length factor that starts at position $i$ of $\boldsymbol{s}$ has a total of $m$ 1's, that is, $|\boldsymbol{s}[i\,..\,i+\ell-1]|_1 = m$.

We formally define the matrix $\mathcal{M}_{\boldsymbol{s}}$ as follows.

**Definition 26** *Given a binary string $\boldsymbol{s}$ of length $n$, $\mathcal{M}_{\boldsymbol{s}}$ is an $n \times n$ matrix such that $\mathcal{M}_{\boldsymbol{s}}[\ell][i] = |\boldsymbol{s}[i\,..\,i+\ell-1]|_1$, for $1 \leq \ell \leq n$ and $1 \leq i \leq (n-\ell+1)$, and $\mathcal{M}_{\boldsymbol{s}}[\ell][i] = 0$, otherwise.*

In what follows, we will use $\mathcal{M}_{\boldsymbol{s}}[\ell]$ to refer to Row $\ell$ of $\mathcal{M}_{\boldsymbol{s}}$. Assume that we are given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ on an alphabet $\Sigma$. For the sake of ease, we assume that (with out loss of generality) $|\boldsymbol{x}| = |\boldsymbol{y}| = n$. We want to find the length of a longest common abelian factor between $\boldsymbol{x}$ and $\boldsymbol{y}$.

**Definition 27** *Given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ over the alphabet $\Sigma$, we say that $\boldsymbol{w}$ is a common abelian factor for $\boldsymbol{x}$ and $\boldsymbol{y}$ if there exist a factor (or substring) $\boldsymbol{u}$ in $\boldsymbol{x}$ and a factor $\boldsymbol{v}$ in $\boldsymbol{y}$ such that $\mathcal{P}_{\boldsymbol{w}} = \mathcal{P}_{\boldsymbol{u}} = \mathcal{P}_{\boldsymbol{v}}$. A common abelian factor of the highest length is called the **Longest Common Abelian Factor (**LCAF**)** between $\boldsymbol{x}$ and $\boldsymbol{y}$. The length of* LCAF *is referred to as the* LCAF ***length**.*

For example, given $A = ababba$ and $B = aaabab$, the LCAF $= 4$. One of such length factor is $abba$ having $\mathcal{P}_{abba} = (2, 2)$.

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | |
| 2 | 1 | 1 | 1 | | |
| 2 | 1 | 2 | | | |
| 2 | 2 | | | | |
| 3 | | | | | |

$M_{A=ababba}$

| | | | | | |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 |
| 2 | 2 | 1 | 1 | 1 | |
| 3 | 2 | 2 | 1 | | |
| 3 | 3 | 2 | | | |
| 4 | 3 | | | | |
| 4 | | | | | |

$M_{B=aaabab}$

In this article, we study the following problem.

**Problem 3.2.1 (**LCAF **Problem)** *Given two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ over the alphabet $\Sigma$, compute the length of an* LCAF *and identify some occurrences of an* LCAF *between $\boldsymbol{x}$ and $\boldsymbol{y}$.*

As has been mentioned in the introduction, trivial brute-force solution of the LCAF problem for a fixed size alphabet has $\mathcal{O}(n^3)$ time and constant space complexity, where $n$ is the length of the longest string. The steps of such an algorithm is given below. Assume that the strings $\boldsymbol{x}$ and $\boldsymbol{y}$ of length $n$ are given.

1: For $\ell \in [1\,..\,n]$
2: For $i \in [1\,..\,n-\ell+1]$
3: Compute the ***Parikh vector*** $\mathcal{P}_{\boldsymbol{x}[i..i+\ell-1]}$
4: For $j \in [1\,..\,n-\ell+1]$
5: Compute the ***Parikh vector*** $\mathcal{P}_{\boldsymbol{y}[j..j+\ell-1]}$
6: If $\mathcal{P}_{\boldsymbol{x}[i..i+\ell-1]}$ is equal to $\mathcal{P}_{\boldsymbol{y}[j..j+\ell-1]}$
7: Then LCAF $= \ell$

Now, suppose that the matrices $\mathcal{M}_{\boldsymbol{x}}$ and $\mathcal{M}_{\boldsymbol{y}}$ for the binary strings $\boldsymbol{x}$ and $\boldsymbol{y}$ have been computed. Now we have the following easy lemma that will be useful for us later.

**Lemma 3.2.2** *There is a common abelian factor of length $\ell$ between $\boldsymbol{x}$ and $\boldsymbol{y}$ if and only if there exists $p, q$ such that $1 \leq p, q \leq n-\ell+1$ and $\mathcal{M}_{\boldsymbol{x}}[\ell][p] = \mathcal{M}_{\boldsymbol{y}}[\ell][q]$.*

*Proof.* [Lemma 3.2.2] Suppose there exists $p, q$ such that $1 \leq p, q \leq n-\ell+1$ and $\mathcal{M}_{\boldsymbol{x}}[\ell][p] = \mathcal{M}_{\boldsymbol{y}}[\ell][q]$. By definition this means $|\boldsymbol{x}[p\,..\,p+\ell-1]|_1 = |\boldsymbol{y}[q\,..\,q+\ell-1]|_1$. So there is a common abelian factor of length $\ell$ between $\boldsymbol{x}$ and $\boldsymbol{y}$. The other way is also obvious by definition. □

Clearly, if we have $\mathcal{M}_{\boldsymbol{x}}$ and $\mathcal{M}_{\boldsymbol{y}}$ we can compute the LCAF by identifying the highest $\ell$ such that there exists $p, q$ having $1 \leq p, q \leq n-\ell+1$ and $\mathcal{M}_{\boldsymbol{x}}[\ell][p] = \mathcal{M}_{\boldsymbol{y}}[\ell][q]$. Then we can say that the LCAF between $\boldsymbol{x}$ and $\boldsymbol{y}$ is the length $\ell$ and common abelian factors of length $\ell$ are $\boldsymbol{x}[p\,..\,p+\ell-1]$ and $\boldsymbol{y}[q\,..\,q+\ell-1]$.

We now generalize the definition of the matrix $\mathcal{M}_{\boldsymbol{s}}$ for strings over a fixed size alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_\sigma\}$ by defining an $n \times n$ matrix $\mathcal{M}_{\boldsymbol{s}}$ of $(\sigma-1)$-length vectors. $\mathcal{M}_{\boldsymbol{s}}[\ell][i] = V_{\ell,i}$, where $V_{\ell,i}[j] = |\boldsymbol{s}[i\,..\,i+\ell-1]|_{\alpha_j}$, for $1 \leq \ell \leq n$, $1 \leq i \leq (n-\ell+1)$ and $1 \leq j < \sigma$, and $V_{\ell,i}[j] = 0$, otherwise. We will refer to the $j$-th element of the array $V_{\ell,i}$ of the matrix $\mathcal{M}_{\boldsymbol{s}}$ by using the notation $\mathcal{M}_{\boldsymbol{s}}[\ell][i][j]$. Notice that the last component of a ***Parikh vector*** is determined by using the length of the string and all the other components of the ***Parikh vector***. Now, $\mathcal{M}_{\boldsymbol{s}}[\ell][i][j] = m$ if and only if the $\ell$-length

factor that starts at position $i$ of $\boldsymbol{s}$ has a total of $m$ $\alpha_j$'s, that is $|\boldsymbol{s}[i\,..\,i+\ell-1]|_{\alpha_j} = m$. Clearly, we can compute $\mathcal{M}_{\boldsymbol{s}}[\ell]$ using the following steps.

1: For $i = 1$ to $n - \ell + 1$ do the following
2: $\quad \mathcal{M}_{\boldsymbol{s}}[\ell][i] = (|\boldsymbol{s}[i\,..\,i+\ell-1]|_{\alpha_1}, \ldots, |\boldsymbol{s}[i\,..\,i+\ell-1]|_{\alpha_{\sigma-1}})$

because we can compute $|\boldsymbol{s}[i+1\,..\,i+1+\ell-1]|_{\alpha_j}$ from $|\boldsymbol{s}[i\,..\,i+\ell-1]|_{\alpha_j}$ in constant time by simply decrementing the $\boldsymbol{s}[i]$ component and incrementing the $\boldsymbol{s}[i+\ell]$ one.

## 3.3 A Quadratic Algorithm

A simple approach for finding the LCAF length considers computing, for $1 \leq \ell \leq n$, the ***Parikh vectors*** of all the factors of length $\ell$ in both $\boldsymbol{x}$ and $\boldsymbol{y}$, i.e., $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$. Then, we check whether $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$ have non-empty intersection. If yes, then $\ell$ could be the LCAF length. So, we return the highest such $\ell$. Moreover, if one knows a ***Parikh vector*** having the LCAF length belonging to such intersection, a linear scan of $\boldsymbol{x}$ and $\boldsymbol{y}$ produces one occurrence of such a factor. The asymptotic time complexity of this approach is $\mathcal{O}(\sigma\, n^2)$ and it requires $\mathcal{O}(\sigma\, n \log n)$ bits of extra space. The basic steps are outlined as follows.

1: For $\ell = 1$ to $n$ do
2: $\quad$ For $i = 1$ to $n - \ell + 1$ do
3: $\quad\quad$ compute $\mathcal{M}_{\boldsymbol{x}}[\ell][i]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell][i]$
4: $\quad$ If $\mathcal{M}_{\boldsymbol{x}}[\ell] \bigcap \mathcal{M}_{\boldsymbol{y}}[\ell] \neq \emptyset$ then
5: $\quad\quad$ LCAF $= \ell$

It is easy to establish that, for fixed length $\ell$, one can compute all the ***Parikh vectors*** in linear time and store them in $\mathcal{O}(\sigma\ n\ \log n)$ bits [Par66]. Now once $\mathcal{M}_{\boldsymbol{x}}$ and $\mathcal{M}_{\boldsymbol{y}}$ are computed, we simply need to apply the idea of Lemma 3.2.2. The idea is to check for all values of $\ell$ whether there exists a pair $p, q$ such that $1 \leq p, q \leq n - \ell + 1$ and $\mathcal{M}_{\boldsymbol{x}}[\ell][p] = \mathcal{M}_{\boldsymbol{y}}[\ell][q]$. Then return the highest value of $\ell$ and corresponding values of $p, q$.

In the binary case, a ***Parikh vector*** is fully represented by just one arbitrary chosen component. Hence, the set of ***Parikh vectors*** of binary factors is just a one dimension list of integers that can be stored in $\mathcal{O}(n \log n)$ bits, since we have $n$ values in the range $[0\,..\,n]$. The intersection can be accomplished in two steps. First, we sort the $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$ rows in $\mathcal{O}(n)$ time by putting them in two lists and using the classic Counting Sort algorithm [CLRS01, Section 8.2]. Then, we check for a non empty intersection with a simple linear scan of the two lists in linear time by starting in parallel from the beginning of the two lists and moving forward element by element on the list having the smallest value among the two examined elements. A further linear scan of $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$ will find the indexes $p, q$ of an element of the not empty intersection. This gives us an $\mathcal{O}(n^2)$ time algorithm requiring $\mathcal{O}(n \log n)$ bits of space for computing an LCAF of two given binary strings.

In the more general case of alphabet greater than two, comparing two ***Parikh vectors*** is no more a constant time operation and checking for empty intersections is not a trivial task. In fact, sorting the set of vectors requires a full order to be defined. We can define an order component by component giving more value to the first component, then to the second one and so on. More formally, we define $\lambda < \mu$, with $\lambda, \mu \in \mathbb{N}^{\sigma}$, if there exist $1 \leq k \leq \sigma$ such that $\lambda[k] < \mu[k]$ and, for any $i$ with $1 \leq i < k$, $\lambda[i] = \mu[i]$. Notice that comparing two vectors will take $\mathcal{O}(\sigma)$ time.

Now, one can sort two lists of $n$ vectors of dimension $\sigma - 1$, i.e., $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$, in $\mathcal{O}(\sigma\ n)$ by using $n$ comparisons taking $\mathcal{O}(\sigma)$ each. Therefore, now the algorithm runs in $\mathcal{O}(\sigma\ n^2)$ time using $\mathcal{O}(\sigma\ n\ \log \sigma)$ bits of extra space.

## 3.4 A Sub-quadratic Algorithm for the Binary Case

In Section 3.3, we have presented an $\mathcal{O}(n^2)$ algorithm to compute the LCAF between two binary strings and two occurrences of common abelian factors, one in each string, having LCAF length. In this section, we show how we can achieve a better running time for the LCAF problem. We will make use of the recent data structure of Moosa and Rahman [MR10] for indexing an abelian pattern. The results of Moosa and Rahman [MR10] is presented in the form of following lemmas with appropriate rephrasing to facilitate our description.

**Lemma 3.4.1** *(Interpolation lemma). If $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are two substrings of a string $\boldsymbol{s}$ on a binary alphabet such that $\ell = |\boldsymbol{s}_1| = |\boldsymbol{s}_2|$, $i = |\boldsymbol{s}_1|_1$, $j = |\boldsymbol{s}_2|_1$, $j > i + 1$, then, there exists another substring $\boldsymbol{s}_3$ such that $\ell = |\boldsymbol{s}_3|$ and $i < |\boldsymbol{s}_3|_1 < j$.*

**Lemma 3.4.2** *Suppose we are given a string $\boldsymbol{s}$ of length $n$ on a binary alphabet. Suppose that $maxOne(\boldsymbol{s}, \ell)$ and $minOne(\boldsymbol{s}, \ell)$ denote, respectively, the maximum and minimum number of $1$'s in any substring of $\boldsymbol{s}$ having length $\ell$. Then, for all $1 \leq \ell \leq n$, $maxOne(\boldsymbol{s}, \ell)$ and $minOne(\boldsymbol{s}, \ell)$ can be computed in $\mathcal{O}(n^2/\log n)$ time and linear space.*

A result similar to Lemma 3.4.1 is contained in the paper of Cicalese et al. [CFL09, Lemma 4], while the result of Lemma 3.4.2 has been discovered simultaneously and independently by Moosa and Rahman [MR10] and by Burcsi et al. [BCFL10]. Notably, in [MR12], a slightly better data structure has been presented which assumes word RAM operations. In addition to the above results we further use the following lemma.

**Lemma 3.4.3** *Suppose we are given two binary strings $\boldsymbol{x}, \boldsymbol{y}$ of length $n$ each. There is a common abelian factor of $\boldsymbol{x}$ and $\boldsymbol{y}$ having length $\ell$ if and only if $maxOne(\boldsymbol{y}, \ell) \geq minOne(\boldsymbol{x}, \ell)$ and $maxOne(\boldsymbol{x}, \ell) \geq minOne(\boldsymbol{y}, \ell)$.*

*Proof.* Assume that $min_{\boldsymbol{x}} = minOne(\boldsymbol{x}, \ell)$, $max_{\boldsymbol{x}} = maxOne(\boldsymbol{x}, \ell)$, $min_{\boldsymbol{y}} = minOne(\boldsymbol{y}, \ell)$, $max_{\boldsymbol{y}} = maxOne(\boldsymbol{y}, \ell)$. Now by Lemma 3.4.1, for all $min_{\boldsymbol{x}} \leq k_{\boldsymbol{x}} \leq max_{\boldsymbol{x}}$, we have some $\ell$-length substrings $A(k_{\boldsymbol{x}})$ of $\boldsymbol{x}$ such that $|\boldsymbol{x}(k_{\boldsymbol{x}})|_1 = k_{\boldsymbol{x}}$. Similarly, for all $min_{\boldsymbol{y}} \leq k_{\boldsymbol{y}} \leq max_{\boldsymbol{y}}$, we have some $\ell$-length factors $\boldsymbol{y}(k)$ of $\boldsymbol{y}$ such that $|\boldsymbol{y}(k_{\boldsymbol{y}})|_1 = k_{\boldsymbol{y}}$. Now, consider the range $[min_{\boldsymbol{x}} \,..\, max_{\boldsymbol{x}}]$ and $[min_{\boldsymbol{y}} \,..\, max_{\boldsymbol{y}}]$. Clearly, these two ranges overlap if and only if $max_{\boldsymbol{y}} \not< min_{\boldsymbol{x}}$ and $max_{\boldsymbol{x}} \not< min_{\boldsymbol{y}}$. If these two ranges overlap then there exists some $k$ such that $min_{\boldsymbol{x}} \leq k \leq max_{\boldsymbol{x}}$ and $min_{\boldsymbol{y}} \leq k \leq max_{\boldsymbol{y}}$. Then we must have some substring $\ell$-length factors $\boldsymbol{x}(k)$ and $\boldsymbol{y}(k)$. Hence the result follows. □

Let us now focus on devising an algorithm for computing the `LCAF` given two binary strings $\boldsymbol{x}$ and $\boldsymbol{y}$ of length $n$. For all $1 \leq \ell \leq n$, we compute $maxOne(\boldsymbol{x}, \ell)$, $minOne(\boldsymbol{x}, \ell)$, $maxOne(\boldsymbol{y}, \ell)$ and $minOne(\boldsymbol{y}, \ell)$ in $\mathcal{O}(n^2/\log n)$ time (Lemma 3.4.2).

Now we try to check the necessary and sufficient condition of Lemma 3.4.3 for all $1 \leq \ell \leq n$ starting from $n$ down to 1. We compute the highest $\ell$ such that

$$[minOne(\boldsymbol{x}, \ell) \,..\, maxOne(\boldsymbol{x}, \ell)] \text{ and } [minOne(\boldsymbol{y}, \ell) \,..\, maxOne(\boldsymbol{y}, \ell)] \text{ overlap.}$$

Suppose that $\mathcal{K}$ is the set of values that is contained in the above overlap, that is

$$\mathcal{K} = \{\, k \mid k \in [minOne(\boldsymbol{x}, \ell) \,..\, maxOne(\boldsymbol{x}, \ell)] \text{ and } k \in [minOne(\boldsymbol{y}, \ell) \,..\, maxOne(\boldsymbol{y}, \ell)] \,\}.$$

Then by Lemma 3.4.3, we must have a set $\mathcal{S}$ of common abelian factors of $\boldsymbol{x}, \boldsymbol{y}$ such that for all $s \in \mathcal{S}$, $|s| = \ell$. Since we identify the highest $\ell$, the length of a longest common factor must be $\ell$, i.e., `LCAF` length is $\ell$. Additionally, we have further identified the number of 1's in such longest factors in the form of the set $\mathcal{K}$. Also, note that for a $k \in \mathcal{K}$ we must have a factor $s \in \mathcal{S}$ such that $|s|_1 = k$.

Now let us focus on identifying an occurrence of the `LCAF`. There are a number of ways to do that. But a straightforward and conceptually easy way is to run the folklore $\ell$-window based algorithm in [MR10] on the strings $\boldsymbol{x}$ and $\boldsymbol{y}$ to find the $\ell$-length factor with number of 1's equal to a particular value $k \in \mathcal{K}$.

The overall running time of the algorithm is deduced as follows. By Lemma 3.4.2, the computation of $maxOne(\boldsymbol{x}, \ell)$, $minOne(\boldsymbol{x}, \ell)$, $maxOne(\boldsymbol{y}, \ell)$ and $minOne(\boldsymbol{y}, \ell)$ can be done in $\mathcal{O}(n^2/\log n)$ time and linear space. The checking of the condition of Lemma 3.4.3 can be done in constant time for a particular value of $\ell$. Therefore, in total, it can be done in $\mathcal{O}(n)$ time. Finally, the folklore algorithm requires $\mathcal{O}(n)$ time to identify an occurrence (or all of them) of the factors. In total the running time is $\mathcal{O}(n^2/\log n)$ and linear space.

## 3.5 Towards a Better Time Complexity

In this section we discuss a simple variant of the quadratic algorithm presented in the previous section. We recall that the main idea of the quadratic solution is to find the greatest $\ell$ with $\mathcal{M}_{\boldsymbol{x}}[\ell] \bigcap \mathcal{M}_{\boldsymbol{y}}[\ell] \neq \emptyset$. The variant we present here is based on the following two simple observations:

1. One can start considering sets of factors of decreasing lengths;
2. When an empty intersection is found between $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$, some rows can possibly be skipped based on the evaluation of the ***gap*** between $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$.

**Algorithm 1** Compute LCAF of $\boldsymbol{x}$ and $\boldsymbol{y}$ using the ***skip trick***.

```
1: function ComputeLCAF(x, y)
2:     set ℓ = n = |x|
3:     while (ℓ ≥ 0) do
4:         par_x = P_{x[1..ℓ]}
5:         par_y = P_{y[1..ℓ]}
6:         for (i = 1; i ≤ (n − ℓ + 1); i++) do
7:             if i == 1 then
8:                 SetMinMax(par_x, par_y)
9:             else
10:                par_x = Slide(par_x, x, ℓ, i)
11:                par_y = Slide(par_y, y, ℓ, i)
12:            push(par_x, list_x)
13:            push(par_y, list_y)
14:            UpdateMinMax(par_x, par_y)
15:        sort list_x, list_y
16:        if list_x ⋂ list_y ≠ ∅ then
17:            return ℓ
18:        ℓ = ℓ− Skip(min_x, max_x, min_y, max_y)
19:    return 0
20: end function
21: function Slide(par, s, ℓ, i)
22:    par[s[i]]- -
23:    par[s[i + l − 1]]++
24:    return par
25: end function
```

```
26: function SKIP(min_x, max_x, min_y, max_y)
27:     gap = 1
28:     for (j = 1; j ≤ σ; j++) do
29:         if min_x[j] > max_y[j] then
30:             tmp = min_x[j] − max_y[j]|
31:         else
32:             tmp = min_y[j] − max_x[j]|
33:         if tmp > gap then
34:             gap = tmp
35:     return gap
36: end function
37: function UPDATEMINMAX(par_x, par_y)
38:     for (c = 1; c ≤ σ; c++) do
39:         if par_x[c] < min_x[c] then
40:             min_x[c] = par_x[c]
41:         if par_y[c] < min_y[c] then
42:             min_y[c] = par_y[c]
43:         if par_x[c] > max_x[c] then
44:             min_x[c] = par_x[c]
45:         if par_y[c] > max_y[c] then
46:             min_y[c] = par_y[c]
47: end function
48: function SETMINMAX(par_x, par_y)
49:     min_x = par_x
50:     min_y = par_y
51:     max_x = par_x
52:     max_y = par_y
53: end function
```

The first observation is trivial. The second observation is what we call the ***skip trick***. Assume that $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$ have been computed and $\mathcal{M}_{\boldsymbol{x}}[\ell] \bigcap \mathcal{M}_{\boldsymbol{y}}[\ell] = \emptyset$ have been found. It is easy to see that, for any starting position $i$ and for any component $j$ (i.e., a letter $a_j$), we have

$$\mathcal{M}_{\boldsymbol{x}}[\ell][i][j] - 1 \leq \mathcal{M}_{\boldsymbol{x}}[\ell-1][i][j] \leq \mathcal{M}_{\boldsymbol{x}}[\ell][i][j] + 1$$

Exploiting this property, we keep track, along the computation of $\mathcal{M}_{\boldsymbol{x}}[\ell]$ and $\mathcal{M}_{\boldsymbol{y}}[\ell]$, of the minimum and maximum values that appear in ***Parikh vectors*** of factors of length $\ell$. We use four arrays indexed by $\sigma$, namely $min_{\boldsymbol{x}}, max_{\boldsymbol{x}}, min_{\boldsymbol{y}}, max_{\boldsymbol{y}}$. Notice that such arrays do not represent ***Parikh vectors*** as they just contain min and max values component by component. Formally, $min_{\boldsymbol{x}}[j] = \min\{\mathcal{M}_{\boldsymbol{x}}[\ell][i][j]\}$, for any $i = 1, .. \ell + 1$. The others have similar definitions.

We compare, component by component, the range of $a_j$ in $\boldsymbol{x}$ and $\boldsymbol{y}$ and we skip as many rows as

$$\max_{j=1}^{\sigma-1}(min_{\boldsymbol{y}}[j] - max_{\boldsymbol{x}}[j]),$$

assuming $min_{\boldsymbol{y}}[j] \geq max_{\boldsymbol{x}}[j]$ (swap $\boldsymbol{x}$ and $\boldsymbol{y}$, otherwise). The modified algorithm is reported in Algorithm 1.

Note that the tricks employed in the modified algorithm are motivated by considering the expected value of LCAF for an independent and identically distributed (i.i.d.) source. It is exponentially close to $n$ according to classic Large Deviation results [Ell85]. The same result is classically extended to ergodic source and it is meant to be a good approximation for real life data when the two strings follow the same probability distribution. Now, we have the following conjecture.

**Conjecture 3.5.1** *The expected length of* LCAF *between two strings* $\boldsymbol{x}$, $\boldsymbol{y}$ *drown from an i.i.d. source is* $\texttt{LCAF}_{avg} = n - \mathcal{O}(\log n)$, *where* $|\boldsymbol{x}| = |\boldsymbol{y}| = n$, *and the number of computed Rows in Algorithm 1 is* $\mathcal{O}(\log n)$ *in average.*

Finally, we will make use of one more trick to speed up the computation of $M$ rows, except the first one, in Algorithm 2. When our algorithm moves from row $M[\ell + 1]$ to row of $M[\ell]$, instead of computing the new row from scratch in $O(n)$ time, we compute the first vector $M[\ell][1]$ of the new row in $O(1)$ time by using the first vector $M[\ell+1][1]$ of the previous computed row. Subsequently, we slide a window of length $\ell$ through the row in $n - \ell$ constant time steps while we compute $M[\ell][j + 1]$, $1 \leq j < n - \ell$. Function *StepDown* in Algorithm 2 is in charge of computing the first ***Parikh vector*** of a new row. To compute $M[\ell][1]$ using $M[\ell+1][1]$, we have to subtract 1 from the vector $M[\ell + 1][1]$ at index $c = \boldsymbol{s}[\ell + 1]$, that is the last character of the factor $w = \boldsymbol{s}[1..\ell + 1]$ of length $\ell + 1$ starting at position 1 in $\boldsymbol{s}$.

For example, consider $\boldsymbol{s} = aacgcctaatcg$, we have $M[12][1] = (4a, 4c, 2g, 2t)$ and $M[11][1] = (4a, 4c, 1g, 2t)$, i.e., $(4a, 4c, 2g, 2t)$ minus $1g$. Then, Function *Slide* in Algorithm 2 computes $M[\ell][j+1]$ from $M[\ell][j]$, for $1 \leq j < n-\ell$, in order to compute the whole row $M[\ell]$. Since we now use the first vector of the previous computed row to compute a new row in $M$, we have to compute first vector even when we skipping some rows. Hence, lines $24 - 29$ of Algorithm 2 compute the first vector $M[\ell][1]$ of row $M[\ell]$ when $\ell$ is a row that is to be skipped.

**Algorithm 2** Compute LCAF of $\boldsymbol{x}$ and $\boldsymbol{y}$ using the ***first vector trick***.

```
1:  function ComputeLCAF(x, y)
2:      set ℓ = n = |x|
3:      first_x = P_{x[1..ℓ]}
4:      first_y = P_{y[1..ℓ]}
5:      while (ℓ ≥ 0) do
6:          par_x = first_x = Stepdown(first_x, x, ℓ)
7:          par_y = first_y = Stepdown(first_y, y, ℓ)
8:          for (i = 1; i ≤ (n − ℓ + 1); i++) do
9:              if i == 1 then
10:                 SetMinMax(par_x, par_y)
11:             else
12:                 par_x = Slide(par_x, x, ℓ, i)
13:                 par_y = Slide(par_y, y, ℓ, i)
14:             push(par_x, list_x)
15:             push(par_y, list_y)
16:             UpdateMinMax(par_x, par_y)
17:         sort list_x, list_y
18:         if list_x ⋂ list_y ≠ ∅ then
19:             return ℓ
20:         skip = Skip(min_x, max_x, min_y, max_y)
21:         while skip > 1 do
22:             ℓ - -
23:             skip - -
24:             first_x = Stepdown(first_x, x, ℓ)
25:             first_y = Stepdown(first_y, y, ℓ)
26:         ℓ - -
27:     return 0
28: end function
```

```
29: function StepDown(par, s, ℓ)
30:     ▷ We assume x and y have a terminal symbol ($).
31:     if s[ℓ] ≠ $ then
32:         par[s[ℓ]] - -
33:     return par
34: end function
```

## 3.6 Experiments

We have conducted some experiments to analyze the behaviour and running time of our skip trick algorithm in practice. The experiments have been run on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. Codes were implemented in $C\#$ language using Visual Studio 2010.

Our first experiment has been carried out principally to verify our rationale behind using the skip trick. We experimentally evaluated the expected number of rows computed in average by using the skip trick of Algorithm 1.

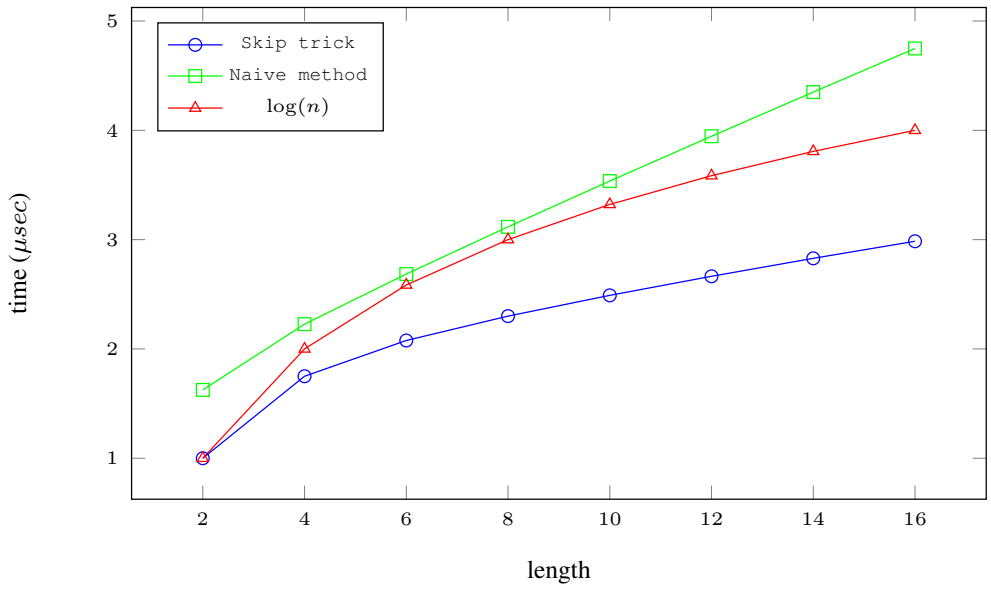



Figure 3.1: Plot of the average number of rows computed executing Algorithm 1 on all the strings of length $2, 3, \ldots 16$ over the binary alphabet.

Figure 3.1 shows the average number of rows computed executing Algorithm 1 on all the strings of length $2, 3, \ldots 16$ over the binary alphabet. Naive method line refers to the number of rows used without the skip trick, starting from $\ell = n$ and decreasing $\ell$ by one at each step. Notice that the skip trick line is always below the $\log n$ line. Figure 3.1 also illustrates that the computed rows, starting from $\ell = n$ to $\ell = n - \log n$, sum up to $\mathcal{O}(\log n)$.

On the other hand, to reach a conclusion in this aspect we would have to increase the value of $n$ in our experiment to substantially more than 64; for $n = 64$, $\sqrt{n}$ is just above $\log n$. Regrettably, limitation of computing power prevents us from doing such an experiment. So, we resort to two more (non-exhaustive) experimental setup as follows to check the practical running time of the skip trick algorithm.

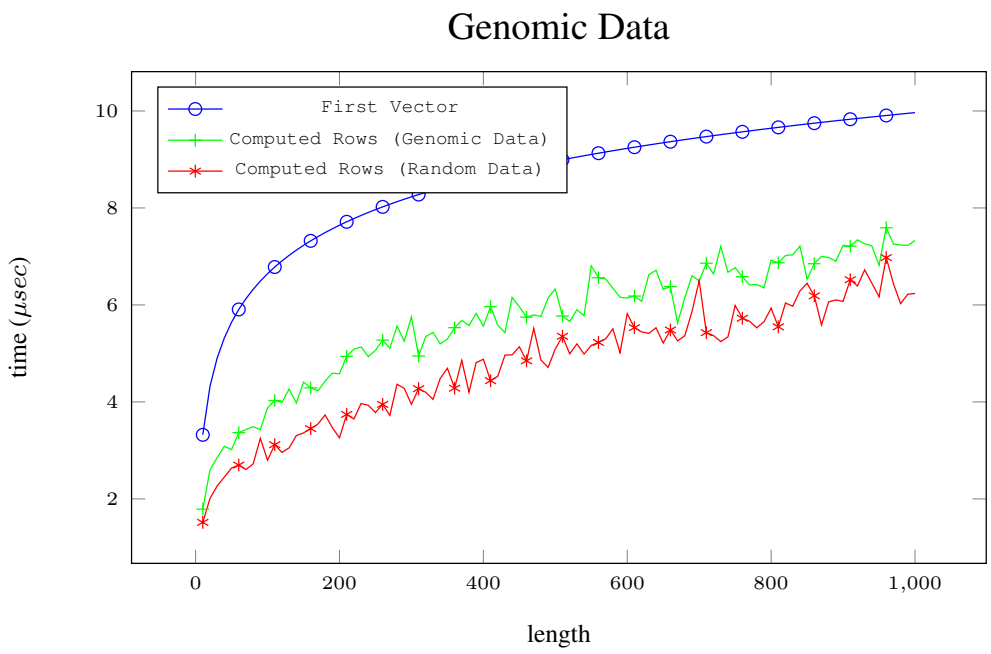



Figure 3.2: Plot of the average number of rows computed executing Algorithm 1 on both genomic and random datasets over the DNA alphabet.

To this end, we conduct our experiments on two datasets, real genomic data and random data. We have taken a sequence (8) from the Homo sapiens genome (250MB) for the former dataset. The latter dataset is generated randomly on the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$). In particular, here we have run the skip trick algorithm on

2 sets of pairs of strings of lengths $10, 20, .., 1000$. For the genomic dataset, these pairs of strings have been created as follows. For each length $\ell, \ell \in \{10, 20, .., 1000\}$ two indexes $i, j \in [1..|\boldsymbol{x}| - \ell]$ have been randomly selected to get a pair of strings $\mathcal{S}[i..i + \ell - 1], \mathcal{S}[j..j + \ell - 1]$, each of length $\ell$. A total of 1000 pairs of strings have been generated in this way for each length $\ell$ and the skip trick algorithm has been run on these pairs to get the average results. On the other hand for random dataset, we simply generate the same number of strings pairs randomly and run the skip trick algorithm on each pair of strings and get the average results for each length group. In both cases, we basically count the numbers of computed rows.

Figure 3.2 shows the average number of rows computed executing Algorithm 1 on both genomic and random datasets over the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$). Notice that the skip trick line is always below the $\log n$ line. Figure 3.2 shows that the computed rows of $\boldsymbol{x}, \boldsymbol{y}$, starting from $\ell = n$ to $\ell = n - \log n$, sum up to $\mathcal{O}(\log n)$.

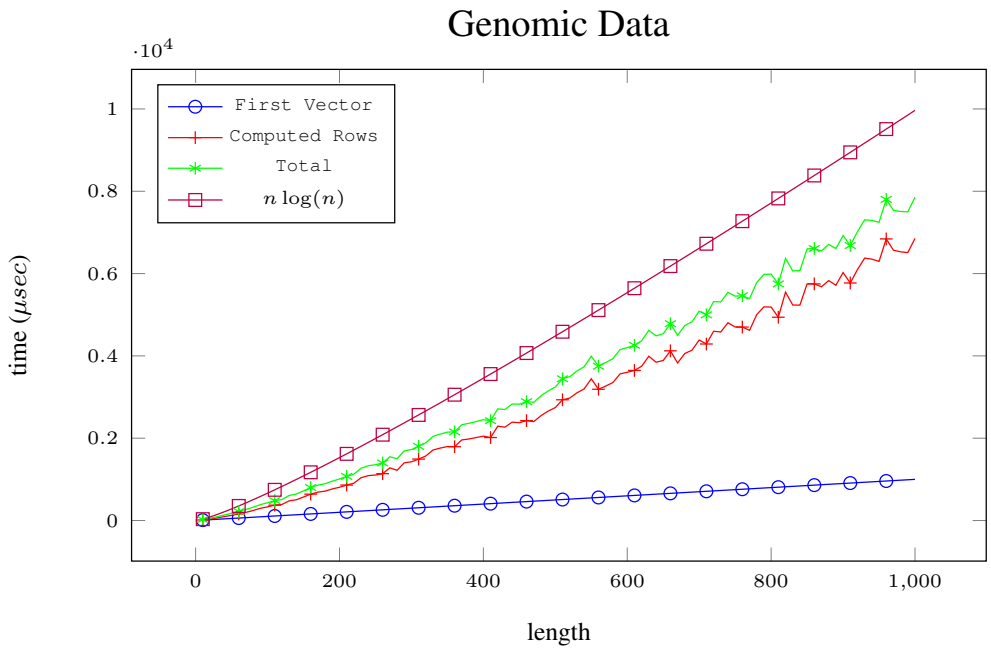



Figure 3.3: Plot of the average number of rows computed executing Algorithm 2 on sequences taken from the Homo sapiens genome.

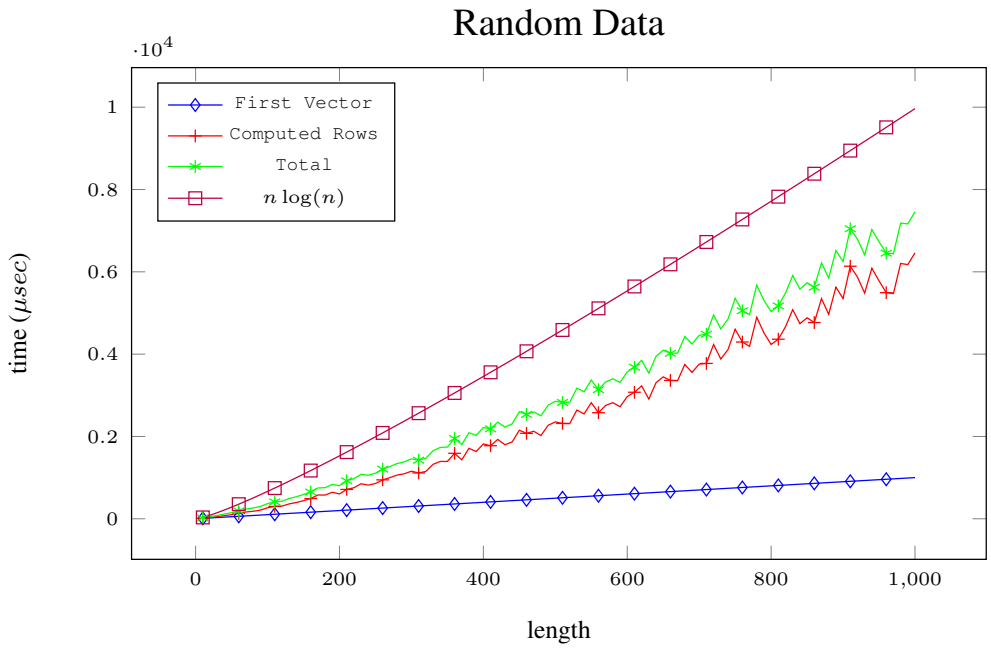



Figure 3.4: Plot of the average number of rows computed executing Algorithm 2 on randomly generated sequences over the alphabet $\Sigma = \{a, c, g, t\}$.

We have further experimentally evaluated the computation of the first vector and the expected number of rows computed in average by employing the first vector trick (Algorithm 2). We have used the same experimental setup as the above. The average number of rows and of the first vector are counted by executing Algorithm 2 on both genomic and random datasets over the DNA alphabet (i.e., $\Sigma = \{a, c, g, t\}$). The results are illustrated in Figures 3.3 & 3.4. In both cases, the figures report the average count of computed rows (Number of Rows), the average count of the first vector (First Vector) and the summation of these two counts (Total). It also shows the $n \log n$ curve. Both of the figures suggest that the algorithm computed the first vector of the visited rows in $\mathcal{O}(n)$ time and the total running time for Algorithm 2 would be $\mathcal{O}(n \log n)$ in practice. Since any row computation takes $\mathcal{O}(\sigma\ n)$, this suggests an average time complexity of $\mathcal{O}(\sigma\ n\ \log n)$, i.e., $\mathcal{O}(n\ \log n)$ for a constant alphabet.

# Article: # 4

# Maximal Palindromic Factorization

A palindrome is a symmetric string, phrase, number, or other sequence of units sequence that reads the same forward and backward.

We present an algorithm for maximal palindromic factorization of a finite string by adapting an Manacher algorithm [Man75] for detecting all occurrences of maximal palindromes in a string in linear time to the length of the given string then using the breadth first search (BFS) to find the maximal palindromic factorization set.

---

## 4.1 Introduction

A palindrome is a symmetric word that reads the same backward and forward. The detection of palindromes is a classical and well-studied problem in computer science, language theory and algorithm design with a lot of variants arising out of different practical scenarios. String and sequence algorithms related to palindromes have long drawn attention of stringology researchers [BG95, Gal76, HCC09, KK09, Man75, ML08, MII+09, PB02]. Interestingly, in the seminal Knuth-Morris-Pratt paper presenting the well-known string matching algorithm [KMP77], a problem related to palindrome recognition was also considered. In word combinatorics, for example, studies have investigated the inhabitation of palindromes in Fibonacci words or Sturmian words in general [Dro95], [DP99], [Gle06].

Manacher discovered an on-line sequential algorithm that finds all *initial* palindromes in a string [Man75]. A string $\boldsymbol{s}[1\,..\,n]$ is said to have an initial palindrome of length $k$ if the prefix $\boldsymbol{s}[1\,..\,k]$ is a palindrome. Gusfield gave a linear-time algorithm to find all *maximal* palindromes (a notion we define shortly) in a string [Gus97]. Porto and Barbosa gave an algorithm to find all *approximate* palindromes in a string [PB02]. Matsubara et al. solved in [MII+09] the problem of finding all palindromes in SLP (Straight Line Programs)-compressed strings. Additionally, a number of problems on variants of palindromes have also been investigated in the literature [HCC09, CHC10, KK09]. Very recently, I *et al.* [ISM11] worked on pattern matching problems and Chowdhury et al. [CHIR14] studied the longest common subsequence problem involving palindromes.

Generic factorization process plays an important role in String Algorithms. The obvious advantage of such process is that when processing a string online, the work done on an element of the factorization can usually be skipped because already done on its previous occurrence [CIS08]. A typical application of this concept resides in algorithms to compute repetitions in strings, such as Kolpakov and Kucherov algorithm for reporting all maximal repetitions [KK99], Lyndon factorization [Mel96], have been applied in: string matching [CP91, BGM11], the Burrows-Wheeler Transform [BW94] and Lempel-Ziv factorization [ZL77] have been applied in: data compression [CDP05, GS12] and indeed it seems to be the only technique that leads to linear-time algorithms independently of the alphabet size [CIS08].

Words with palindromic structure are important in DNA and RNA sequences, Biologists believe that palindromes play an important role in regulation of gene activity and other cell processes because these are often observed near promoters, introns and specific untranslated regions. Palindromic structure in DNA and RNA sequences reflects the capacity of molecules to fold [KK09], i.e. to form double-stranded stems, which insures a stable state of those molecules with low free energy. Identifying palindromes could help in advancing the understanding of genomic instability [Cho05b], [LSvD+09], [TBYT06]. Finding common palindromes in two gene sequences can be an important criterion to compare them, and also to find common relationships between them. However, in those applications, the reversal of palindromes should be combined with the complementarity concept on nucleotides, where $c$ is complementary to $g$ and $a$ is complementary to $t$ (or to $u$, in case of RNA). Moreover, gapped palindromes are biologically meaningful, i.e. contain a spacer between left and right copies (see [KK09]).

Therefore, detecting palindromes in DNA sequences is one of the challenging problems in computational biology. Researchers have also shown that based on palindrome frequency, DNA sequences can be discriminated to the level of species of origin [LBLM11]. So, finding common palindromes in two DNA sequences can be an important criterion to compare them, and also to find common relationships between them.

## 4.2 Notations and terminology

A palindrome is a symmetric string that reads the same forward and backward. More formally, $s$ is called a palindrome if and only if $\boldsymbol{s} = \tilde{\boldsymbol{s}}$, where $\tilde{\boldsymbol{s}}$ is the reverse of $\boldsymbol{s}$. The empty string $\epsilon$ is assumed to be a palindrome. Also note that a single character is a palindrome by definition. The following is another (equivalent) definition of a palindrome which indicates that palindrome can be of both odd and even length. A string $\boldsymbol{s}$ is a palindrome if $\boldsymbol{s} = \boldsymbol{u}a\tilde{\boldsymbol{u}}$ where $\boldsymbol{u}$ is a string and $a$ is either a single character or the empty string $\epsilon$. Clearly, if $a$ is a single character, then $\boldsymbol{s}$ is a palindrome having odd length; otherwise, it is of even length. The radius of a palindrome $\boldsymbol{s}$ is $\frac{|\boldsymbol{s}|}{2}$. In the context of a string, if we have a substring that is a palindrome, we often call it a palindromic substring. Given a string $\boldsymbol{s}$ of length $n$, suppose $\boldsymbol{s}[i\,..\,j]$,with $1 \leq i \leq$

$j \leq n$ is a palindrome, i.e., $\boldsymbol{s}[i\,..\,j]$ is a palindromic substring of $\boldsymbol{s}$. Then, the center of the palindromic substring $\boldsymbol{s}[i\,..\,j]$ is $\frac{i+j}{2}$. A palindromic substring $\boldsymbol{s}[i\,..\,j]$ is called the maximal palindrome at the center $\frac{i+j}{2}$ if no other palindromes at the center $\frac{i+j}{2}$ have a larger radius than $\boldsymbol{s}[i\,..\,j]$, i.e., if $\boldsymbol{s}[i-1] \neq \boldsymbol{s}[j+1]$. A maximal palindrome $\boldsymbol{s}[i\,..\,j]$ is called a suffix (prefix resp.) palindrome of $\boldsymbol{s}$ if and only if $j = n$ ($i = 1$ resp.). We denote by $(c, r)_{\boldsymbol{s}}$ the maximal palindromic factor of a string $\boldsymbol{s}$ whose center is $c$ and radius is $r$; we usually drop the subscript and use $(c, r)$ when the string $\boldsymbol{s}$ is clear from the context. The set of all center-distinct maximal palindromes of a string $\boldsymbol{s}$ is denoted by $\mathcal{MP}(\boldsymbol{s})$. Further, for the string $\boldsymbol{s}$, we denote the set of all *prefix palindromes* (*suffix palindromes*) as $\mathcal{PP}(\boldsymbol{s})$ ($\mathcal{SP}(\boldsymbol{s})$). We use the following result from [Man75, Gus97].

**Theorem 4.2.1 ([Man75, Gus97])** *For any string $\boldsymbol{s}$ of length $n$, $\mathcal{MP}(\boldsymbol{s})$ can be computed in $\mathcal{O}(n)$ time.*

In what follows, we assume that the elements of $\mathcal{MP}(\boldsymbol{s})$ are sorted in increasing order of centers $c$. Actually, the algorithm of [Man75] computes the elements of $\mathcal{MP}(\boldsymbol{s})$ in this order. Clearly, the set $\mathcal{PP}(\boldsymbol{s})$ and $\mathcal{SP}(\boldsymbol{s})$ can be computed easily during the computation of $\mathcal{MP}(\boldsymbol{s})$.

Suppose, we are given a set of strings $\mathcal{S} = \{\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots, \boldsymbol{s}_k\}$, such that $\boldsymbol{s}_i$ is a substring of $\boldsymbol{s}$ and $1 \leq i \leq k$. A factorization $\mathcal{F}$ of $\boldsymbol{s}$ with respect to $\mathcal{S}$ refers to a decomposition of $\boldsymbol{s}$ such that $s = s_{i_1} s_{i_2} \ldots s_{i_\ell}$ where $s_{i_j} \in \mathcal{S}$ and the number of the factors $\ell$ is minimum. In this context the set $\mathcal{S}$ is referred to as the factorization set. In this article, we tackle the following problem.

**Problem 4.2.2 (Maximal Palindromic Factorization (MPF))** *Given a string $\boldsymbol{s}$, find the maximal palindromic factorization of $\boldsymbol{s}$, that is a factorization of $\boldsymbol{s}$ where the factorization set is $\mathcal{MP}(\boldsymbol{s})$.*

Sometimes `MPF` doesn't produce any factorization for example $abaca$. Even when it produces factorization, it may consist of more than the minimum number of palindromic (maximal or not) substrings into which the string can be factored, for example $abbaabaabbba$ can be factored into $abba$, $aba$ and $abbba$ but cannot be factored into fewer than four maximal palindromic substrings.

## 4.3 The Algorithm

In this section we present an algorithm to compute the maximal palindromic factorization of a given string $\boldsymbol{s}$. We first present some notions required to present our algorithm. First of all, recall that we use $\mathcal{MP}(\boldsymbol{s})$ to denote the set of center distinct maximal palindromes of $\boldsymbol{s}$. We further extend this notation as follows. We use $\mathcal{MP}(\boldsymbol{s})[i]$, where $1 \leq i \leq n$ to denote the set of maximal palindromes with center $i$.

**Proposition 4.3.1** *The position $i$ could be the center of at most two maximal palindromic factors, therefore; $\mathcal{MP}(\boldsymbol{s})[i]$ contains at most two elements, where $1 \leq i \leq n$, hence; there are at most $2n+1$ elements in $\mathcal{MP}(\boldsymbol{s})$.*

On the other hand, we use $\mathcal{MPL}(\boldsymbol{s})[i]$ to denote the set of the lengths of all maximal palindromes ending at position $i$,where $1 \leq i \leq n$ in $\boldsymbol{s}$.

$$\mathcal{MPL}(\boldsymbol{s})[i] = \{2\ell - 1 \mid \boldsymbol{s}[i-\ell+1\,..\,i+\ell-1] \in \mathcal{MP}(\boldsymbol{s})\} \cup \{2\ell' \mid \boldsymbol{s}[i-\ell'\,..\,i+\ell'-1] \in \mathcal{MP}(\boldsymbol{s})\} \quad (4.1)$$

where $1 \leq i \leq n$, with $2\ell$ and $2\ell'+1$ are the lengths of the odd and even palindromic factors respectively.

**Proposition 4.3.2** *The set $\mathcal{MPL}(\boldsymbol{s})$ (Equation 4.1) can be computed in linear time from the set $\mathcal{MP}(\boldsymbol{s})$.*

Now we define the list $\mathcal{U}(\boldsymbol{s})$ such that for each $1 \leq i \leq n$, $\mathcal{U}(\boldsymbol{s})[i]$ stores the position $j$ such that $j+1$ is the starting position of a maximal palindromic factors ending at $i$ and $j$ is the end of another maximal palindromic substring.

Clearly, this can be easily computed once we have $\mathcal{MPL}(\boldsymbol{s})$ computed.

$$\mathcal{U}[i][j] = i - \mathcal{MPL}(\boldsymbol{s})[i][j] \quad (4.2)$$

One can observe, from Proposition 4.3.1, that the sets $\mathcal{MPL}(\boldsymbol{s})$ and $\mathcal{U}(\boldsymbol{s})$ contain at most $2n+1$ elements.

Given the list $\mathcal{U}(\boldsymbol{s})$ for a string $\boldsymbol{s}$, we define a directed graph $\mathcal{G}_{\boldsymbol{s}} = (\mathcal{V}, \mathcal{E})$ as follows. We have $\mathcal{V} = \{i \mid 1 \leq i \leq n\}$ and $\mathcal{E} = \{(i, j) \mid j \in \mathcal{U}(\boldsymbol{s})[i]\}$. Note that $(i, j)$ is a directed edge where the direction is from $i$ to $j$. Now we can present the steps of our algorithm for computing the maximal palindromic factorization of a given string $\boldsymbol{s}$ of length $n$. The steps are as follows.

`MPF` **Algorithm: Maximal Palindromic Factorization Algorithm**
**Input:** A String $\boldsymbol{s}$ of length $n$
**Output:** Maximal Palindromic Factorization of $\boldsymbol{s}$

1: Compute the set of maximal palindromes $\mathcal{MP}(\boldsymbol{s})$ and identify the set of prefix palindromes $\mathcal{PP}(\boldsymbol{s})$.
2: Compute the list $\mathcal{MPL}(\boldsymbol{s})$.
3: Compute the list $\mathcal{U}(\boldsymbol{s})$.
4: Construct the graph $\mathcal{G}_{\boldsymbol{s}} = (\mathcal{V}, \mathcal{E})$.
5: Do a breadth first search on $\mathcal{G}_{\boldsymbol{s}}$ assuming the vertex $n$ as the source.
6: Identify the shortest path $P \equiv n \rightsquigarrow v$ such that $v$ is the end position of a palindrome belonging to $\mathcal{PP}(\boldsymbol{s})$. Suppose $P \equiv \langle n = p_k\ p_{k-1}\ ..\ p_2\ p_1 = v \rangle$.
7: Return $\boldsymbol{s} = \boldsymbol{s}[1\,..\,p_1]\boldsymbol{s}[p_1 + 1\,..\,p_2]\ ..\ \boldsymbol{s}[p_{k-1} + 1\,..\,p_k]$.

## 4.4 Analysis

CORRECTNESS:
We now have the following theorem which proves the correctness of `MPF` Algorithm.

**Theorem 4.4.1 (Correctness and Running time)** *Given a string $\boldsymbol{s}$ of length $n$,* `MPF` *Algorithm correctly computes the maximal palindromic factorization of $\boldsymbol{s}$ in $\mathcal{O}(n)$ time.*

*Proof.* We first focus on an edge $(i, j) \in \mathcal{E}$ of the graph $\mathcal{G}_{\boldsymbol{s}}$ constructed at Step 4 of the algorithm. By definition, this means the following:

1. There is a maximal palindrome $pal_i$ having length $\ell_i$ (say) ending at position $i$.
2. There is a maximal palindrome $pal_j$ having length $\ell_j$ (say) ending at position $j$.

3. $i > j$.

4. $i - \ell_i = j$.

Since, by definition, each directed edge $(i, j) \in \mathcal{E}$ is such that $i > j$, so, for a path $P \equiv \langle p_k\ p_{k-1}\ ..\ p_2\ p_1 \rangle$ in $\mathcal{G}_{\boldsymbol{s}}$, we always have $p_k > p_{k-1} > \ .. \ > p_1$. A path $P \equiv \langle p_k\ p_{k-1}\ ..\ p_2\ p_1 \rangle$ can be seen as corresponding to a substring of $\boldsymbol{s}$ formed by concatenation of maximal palindromes as follows.

Each edge $(p_i, p_{i-1}) \in P$ corresponds to a palindromic substring $\boldsymbol{s}[p_{i-1}]\boldsymbol{s}[p_{i-1} + 1]\boldsymbol{s}[p_{i-1} + 2] \,..\, \boldsymbol{s}[p_i]$.

Hence, following the definition of the edges, it is clear that any path would correspond to a substring of $\boldsymbol{s}$ formed by concatenation of consecutive palindromic substrings.

In Step 5, a breadth first tree is constructed from $\mathcal{G}_{\boldsymbol{s}}$ considering the vertex $n$ as the source. A breadth first tree gives the shortest path from the source (in this case, $n$) to any other node. Now, in Step 6, `MPF` Algorithm identifies the set of shortest paths (say, $SPath$) between $n$ and $j$, such that $j$ corresponds to a maximal palindromic prefix of $\boldsymbol{s}$. Now the maximum palindromic factorization must contain exactly one palindrome from $\mathcal{PP}(\boldsymbol{s})$ and exactly one palindrome from $\mathcal{SP}(\boldsymbol{s})$, where the length of the shortest path is minimum. Hence, it is easy to realize that the shortest one among the paths in $SPath$ corresponds to the maximal palindromic factorization. This completes the correctness proof.

### RUNNING TIME:

In Step 1 the computation of $\mathcal{MP}(\boldsymbol{s})$ can be done using the algorithm of [Man75] in $\mathcal{O}(n)$ time. Also, $\mathcal{PP}(\boldsymbol{s})$ and $\mathcal{SP}(\boldsymbol{s})$ can be computed easily while computing $\mathcal{MP}(\boldsymbol{s})$. The computation of $\mathcal{MPL}(\boldsymbol{s})$ and $\mathcal{U}(\boldsymbol{s})$ in Step 2 and Step 3 can be done in linear time once $\mathcal{MP}(\boldsymbol{s})$ is computed.

Now construction of the graph $\mathcal{G}_{\boldsymbol{s}}$ is done in Step 4. There are in total $n$ number of vertices is $\mathcal{G}_{\boldsymbol{s}}$. The number of edges $|\mathcal{E}|$ of $\mathcal{G}_{\boldsymbol{s}}$ depends on $\mathcal{U}(\boldsymbol{s})$. But it is easy to realize that the summation of the number of elements in all the positions of $\mathcal{U}(\boldsymbol{s})$ cannot exceed the total number of maximal palindromes. Now, since there can be at most $2n + 1$ centers, there can be just as many maximal palindromes in $\boldsymbol{s}$. Therefore we have $|\mathcal{E}| = \mathcal{O}(n)$.

Hence, the graph construction (Step 4) as well as the breadth first search (Step 5) can be done in $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|) = \mathcal{O}(n)$ time. Finally, the identification of the desired path in Step 6 can also be done easily if we do some simple book-keeping during the breadth first search. Next we will discuss the implementation steps for completeness.

BFS algorithm (see 1.3.1 for more details) is to traverse the graph as close as possible to the root node. Starting from the source node, the nodes at distance 1 from it, the nodes at distance 2 from it, and so on.

Vertices are implemented as linked list. Two vertices are adjacent when they are both incident to a common edge. A path is a sequence of vertices $P = (v_1, v_2, .. v_n)$ in a graph such that $v_i$ and $v_{i+1}$ are adjacent for $i \leq i \leq n$. The length of the path is the number of edges traversed. Edges represent the connection between nodes. There are two ways to represent edges, adjacency matrix or adjacency list representation of graphs and queue of nodes is used in the implementation of the breadth first search. Elements are extracted in first-in-first-out (FIFO) order, i.e., elements are picked in the order in which they were inserted. Each vertex will enter the queue once. Then the visited state will be set to `true`, and after the vertex is de-queued it can never enter the queue again.

Nodes are implemented as objects that store: current vertex ID, predecessor node, path length and back-pointers to other nodes.

In order to reconstruct the path to the source vertex after the destination vertex has been found, some book-keeping has to be done as the search progresses.

One approach would be maintaining a mapping from each node to its parent, and when inspecting the adjacent node, record its parent. This map will be populated during the iterations of BFS. For each vertex, it contains a reference to the vertex through which the BFS entered that vertex. If we follow these references back, once the search is done, we will arrive at the vertex the BFS started at. Thus for every vertex we can reconstruct the path from the start vertex to the current vertex that the BFS took.

Note that the logical parents map can also be implemented as an array, if the vertices are enumerated and we can reconstruct the path by simply going from the target node up until we get back to the source node.

Since we already have computed the sets $\mathcal{PP}(\boldsymbol{s})$ and $\mathcal{SP}(\boldsymbol{s})$ in Step 1. Hence the total running time of the algorithm is $\mathcal{O}(n)$. And this completes the proof. □

### 4.4.1 An Illustrative Example

Suppose we are given a string $\boldsymbol{s} = addcbbcbbbcbb$. We will proceed as follows:

First we compute the set $\mathcal{MP}(\boldsymbol{s})$. For example, there is a palindrome of length $9$ centered at position $9$ of $s$ and at position $i = 7$ there is a palindrome of length $5$ centered at position $7$ of $\boldsymbol{s}$ .

Secondly, we compute the set $\mathcal{MPL}(\boldsymbol{s})$. For example, at position $i = 9$ there are $2$ palindromes of lengths $2$ and $5$ ending at position $9$ of $\boldsymbol{s}$.

Finally, we compute $\mathcal{U}(\boldsymbol{s})$ (Table 4.1 shows full steps for $\boldsymbol{s} = addcbbcbbbcbb$).

Now, we can construct the graph $\mathcal{G}_{\boldsymbol{s}}$ easily as shown in Figure 4.1. For example, we can see that from vertex $i = 9$ we have $2$ directed edges, namely, $(9, 7)$ and $(9, 4)$. Our desired shortest path is $P = \langle 13, 4, 3, 1\rangle$ (corresponding edges are shown as dashed edges). So, the maximal palindromic factorization of $\boldsymbol{s} = addcbbcbbbcbb$ is as follows:

$$s[1..1]s[2..3]s[4..4]s[5..13] = a\ dd\ c\ bbcbbbcbb$$

.

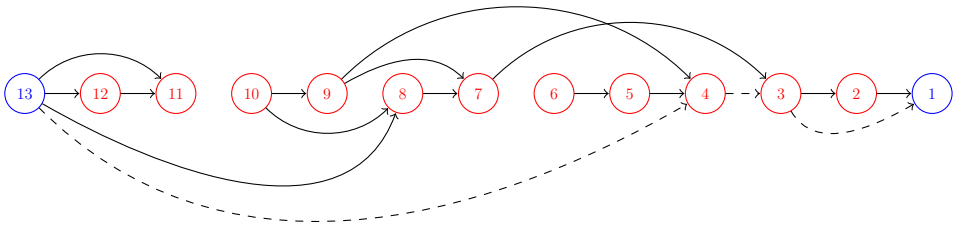



Figure 4.1: The graph $\mathcal{G}_s$ for $s = addcbbcbbbcbb$

Table 4.1: Steps for computing $\mathcal{U}(\boldsymbol{s})$ and $\mathcal{MPL}(\boldsymbol{s})$ for $\boldsymbol{s} = addcbbcbbbcbb$

| $i$ | $\mathcal{MPL}[i]$ | $\mathcal{U}[i] = i - \mathcal{MPL}[i]$ |
|---|---|---|
| 1 | $\mathcal{MPL}[1] = \{1\}$ | $\mathcal{U}[1] = \{0\}$ |
| 2 | $\mathcal{MPL}[2] = \{1\}$ | $\mathcal{U}[2] = \{1\}$ |
| 3 | $\mathcal{MPL}[3] = \{1, 2\}$ | $\mathcal{U}[3] = \{2, 1\}$ |
| 4 | $\mathcal{MPL}[4] = \{1\}$ | $\mathcal{U}[4] = \{3\}$ |
| 5 | $\mathcal{MPL}[5] = \{1\}$ | $\mathcal{U}[5] = \{4\}$ |
| 6 | $\mathcal{MPL}[6] = \{1\}$ | $\mathcal{U}[6] = \{5\}$ |
| 7 | $\mathcal{MPL}[7] = \{4\}$ | $\mathcal{U}[7] = \{3\}$ |
| 8 | $\mathcal{MPL}[8] = \{1\}$ | $\mathcal{U}[8] = \{7\}$ |
| 9 | $\mathcal{MPL}[9] = \{2, 5\}$ | $\mathcal{U}[9] = \{7, 4\}$ |
| 10 | $\mathcal{MPL}[10] = \{1, 2\}$ | $\mathcal{U}[10] = \{9, 8\}$ |
| 11 | $\mathcal{MPL}[11] = \{.\}$ | $\mathcal{U}[11] = \{.\}$ |
| 12 | $\mathcal{MPL}[12] = \{1\}$ | $\mathcal{U}[12] = \{11\}$ |
| 13 | $\mathcal{MPL}[13] = \{1, 2, 5, 9\}$ | $\mathcal{U}[13] = \{12, 11, 8, 4\}$ |

## 4.5 Maximal Biological palindromic factorization

DNA in its natural, double-stranded form may contain palindromes, sequences which read the same from either side because they are identical to their reverse complement on the sister strand. It is not surprising that biological palindromes are of particular importance in genome biology due to the fact that they are the only sequences which are unique in double-stranded DNA.

**Definition 28** *The following definition from [Gus97] complemented (biological)palindromes can refer to back-to-back DNA segments, that is, a DNA or RNA string that becomes a palindrome if each character in one half of the string is changed to its complement character in DNA, A - T are complements and C - G are complements; in RNA A - U and C - G are complements). For example,* $GAATTC$, $AAGCTT$, $GCCCGGGC$ *and* $AGCTCGCGAGCT$ *are a complemented palindrome.*

The algorithm presented in Section 4.3 can be extended to biological palindromes, where the word reversal is defined in conjunction with the complementarity of nucleotide letters: $c \leftrightarrow g$ and $a \leftrightarrow t$ (or $a \leftrightarrow u$, in case of RNA). The main part of the algorithm is extended in a straightforward way: when computing $\mathcal{MP}(\boldsymbol{s})$ one has to use the complementarity relation, i.e., each time the algorithm compares two letters, this comparison is replaced by testing their complementarity.

# Article: # 5

# Lyndon Fountains and the Burrows-Wheeler Transform

In this article, we study Lyndon structures related to the Burrows-Wheeler Transform with potential application to bioinformatics. Next-Generation Sequencing techniques require the alignment of a large set of short reads (between dozens to hundreds of letters) on a reference sequence (millions of letters). The Burrows-Wheeler Transform has been used in various alignment programs which generally compute the Lyndon factorization of the reference sequence as a preprocessing step. We compute the quadratic factorization of all rotations of an input string and the Burrows-Wheeler Transform of a Lyndon substring. From the factored rotations we introduce the *Lyndon fountain*.

---

## 5.1 Introduction

A Lyndon word is defined as a (generally) finite word which is minimal for the lexicographic order of its conjugacy class; the set of Lyndon words permits the unique maximal factorization of any given string [CFL58, Lot83]. Lyndon words have been applied in: string matching [CP91, BGM11], the Burrows-Wheeler Transform [BW94] and data compression [CDP05, GS12], musicology [Che04], bioinformatics [DR04], cryptanalysis [Per05], string combinatorics [Duv83, Smy03], Free Lie algebras [Reu93]. Our focus here is novel combinatorial properties of Lyndon words with potential applications to bioinformatics in the context of Next-Generation Sequencing (NGS) techniques. In NGS, large unknown DNA sequences are fragmented into small segments (a few dozens to several hundreds of base pairs long). This generates masses of data, typically several million "short reads". In order to reconstruct the original DNA sequence, alignment programs attempt to align or match these reads to a reference genome. Alignment programs first used hashing or the suffix tree/array data structures; subsequently, efficiency in memory requirement was achieved by using the compression related Burrows-Wheeler Transform (BWT) [ABM08, SLLM09]. Implementations based on the BWT include: SOAP2 [LYL+09], BWA [LD09], and Bowtie [LTPS09]. Further, word-level parallelism has been applied to NGS technologies [ADI+09]. The input to the BWT is a reference genome, comprising in the case of the human genome of about 3 billion DNA base pairs (from {A,C,G,T}). Space saving techniques with the BWT are achieved by first factoring the string into Lyndon words [Duv83]. In this note we study structural properties of the factorization of a set of rotations of a Lyndon word. In particular, we propose algorithms for factoring this set of rotations, and for obtaining the BWT of a Lyndon sub-word on the fly while computing the BWT of a Lyndon word.

## 5.2 Burrows-Wheeler Transform

As described in [BW94] the algorithm transforms a string $\boldsymbol{T}$ (text) of $n$ characters by forming the $n$ rotations (cyclic shifts) of $\boldsymbol{T}$, sorting them lexicographically, and extracting the last character of each of the rotations. A string $\boldsymbol{L}$ (last) is formed from these characters, where the $i$-th character of $\boldsymbol{L}$ is the last character of the $i$-th sorted

rotation. In addition to $\boldsymbol{L}$, the algorithm computes the index $I$ of the original string $\boldsymbol{T}$ in the sorted list of rotations. Given only $\boldsymbol{L}$ and $I$, there is an efficient algorithm [BW94] to compute the original string $\boldsymbol{T}$. The transformed text $\boldsymbol{T}^{BWT}$ is the last column $\boldsymbol{L}$ (last). Notice that every row and every column of $M$, hence also the transformed text $\boldsymbol{L}$ is a permutation of $\boldsymbol{T}$. In particular the first column of $M$, call it $\boldsymbol{F}$ (first), is obtained by lexicographically sorting the characters of $\boldsymbol{T}$ (or, equally, the characters of $\boldsymbol{L}$). The transformed string $L$ usually contains long runs of identical symbols and therefore can be efficiently compressed using a simple locally-adaptive compression algorithm (see Figure. 5.1 for example).

$$
\begin{bmatrix}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 & F & & & & & & L \\
1 & B & A & N & A & N & A & \$ \\
2 & A & N & A & N & A & \$ & B \\
3 & N & A & N & A & \$ & B & A \\
4 & A & N & A & \$ & B & A & N \\
5 & N & A & \$ & B & A & N & A \\
6 & A & \$ & B & A & N & A & N \\
7 & \$ & B & A & N & A & N & A
\end{bmatrix}
\overset{sort}{\Rightarrow}
\begin{bmatrix}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 & F & & & & & & L \\
7 & \$ & B & A & N & A & N & A \\
6 & A & \$ & B & A & N & A & N \\
4 & A & N & A & \$ & B & A & N \\
2 & A & N & A & N & A & \$ & B \\
1 & B & A & N & A & N & A & \$ \\
5 & N & A & \$ & B & A & N & A \\
3 & N & A & N & A & \$ & B & A
\end{bmatrix}
$$

Figure 5.1: BWT example, using $\boldsymbol{T} = BANANA$. First, we append a unique (end-of-string) symbol \$ (lexicographically the smallest character in $\Sigma$) to $\boldsymbol{T}$ to get $\boldsymbol{T}' = BANANA\$$, and consider all its rotations, then we sort these rotations to obtain the BWT matrix. By taking the last column, we get the $\boldsymbol{T}^{BWT}$: $ANNB\$AA$.

## 5.3 Lyndon fountain

Let $\mathcal{L}$ be the set of Lyndon words over an alphabet $\Sigma$. For $\boldsymbol{\ell} \in \mathcal{L}$, where $\boldsymbol{\ell} = \ell_1\ell_2...\ell_n$, let $\mathcal{R}(\boldsymbol{\ell})$ be the set of $n$ cyclic rotations of $\boldsymbol{\ell}$ (to avoid trivialities we assume that $n > 1$). These rotations are organized as an $n \times n$ matrix: the first row $r_1$ in the matrix is the Lyndon word $\ell$, and the $i$-th row, $r_i$, is the $i$-th rotation. If row $\boldsymbol{r_i} \in \mathcal{R}(\boldsymbol{\ell})$ and $\boldsymbol{r_i} = r_1r_2...r_n$ (with $r_i$ in position $i$), we cycle clock-wise, that is, $\boldsymbol{r_{i+1}} = r_nr_1r_2...r_{n-1}$. Consider the set (matrix) $\mathcal{R}(\boldsymbol{\ell})^<$ of cyclic rotations of $\boldsymbol{\ell}$ ordered in lexicographic order.

The BWT is the last right hand column of $\mathcal{R}(\boldsymbol{\ell})^{<}$. We first give some results related to factoring every row of $\mathcal{R}(\boldsymbol{\ell})$, and then introduce the concept of the *Lyndon fountain*, followed by outlining the algorithms.

For a string $\boldsymbol{s}$, let $f(\boldsymbol{s})$ be the Lyndon factorization of $\boldsymbol{s}$. Further, let $F(\mathcal{R}(\boldsymbol{\ell}))$ be the set of Lyndon factorizations of $\mathcal{R}(\boldsymbol{\ell})$, that is, the set $\{f(\boldsymbol{r_i}),\ 1 \leq i \leq n\}$. The first row $f(\boldsymbol{r_1})$ in $F(\mathcal{R}(\boldsymbol{\ell}))$ is $f(\boldsymbol{r_1}) = f(\boldsymbol{\ell}) = (\ell_1\ell_2...\ell_n)$, and from Lyndon properties we have $\ell_1 < \ell_n$. Thus $f(\boldsymbol{r_2}) = f(\ell_n\ell_1\ell_2...\ell_{n-1}) = (\ell_n)f(\ell_1\ell_2...\ell_{n-1})$, that is, the factorization commences with the unit factor (a factor of length one) $(\ell_n)$. Similarly, since each factor in $f(\boldsymbol{r_n})$ starts with a letter $\ell \geq \ell_1$, the factorization of the last row ends with the unit factor $(\ell_1)$. The *diagonal* of $F(\mathcal{R}(\boldsymbol{\ell}))$ consists of the sequence of Lyndon factors $\boldsymbol{\ell}_i$, $1 \leq i \leq n$, such that $\boldsymbol{\ell}_i$ starts at position $i$ in $f(\boldsymbol{r_i})$; hence each $\boldsymbol{\ell}_i$ starts with the letter $\ell_1$. We now show that no Lyndon word in $F(\mathcal{R}(\boldsymbol{\ell}))$ crosses the diagonal, and any Lyndon factors adjacent to but on opposite sides of the diagonal are distinct:

**Observation 5.3.1** *Let $\boldsymbol{\ell}$ be a Lyndon word ($\boldsymbol{\ell} \in \mathcal{L}$) where $\boldsymbol{\ell} = \ell_1\ell_2\,..\,\ell_n$ (to avoid trivialities we assume $n > 1$), then*

*(i)* $border(\boldsymbol{\ell}) = 0$,
*(ii)* $\ell_1 < \ell_n$,
*(iii)* $\ell_1 \leq \ell_i | \ell_i \in \{\ell_2, .., \ell_{n-1}\}$,
*(iv)* $\boldsymbol{\ell} < \ell_i \cdots \ell_n$, *for* $1 < i \leq n$.

**Lemma 5.3.2** *Let $\boldsymbol{\ell} = \ell_1\ell_2...\ell_n \in \mathcal{L}$, and let $\boldsymbol{\ell}', \boldsymbol{\ell}'' \in F(\mathcal{R}(\boldsymbol{\ell}))$.*

*(i) Suppose $\boldsymbol{\ell}' \in \boldsymbol{r_i}$ and $\boldsymbol{\ell}' = \ell_s\ell_{s+1}...\ell_t$. If $\ell_s$ is in position $j$, $j < i$ then the position $k$ of $\ell_t$ is such that $k < i$.*
*(ii) If $\boldsymbol{\ell}' = \ell_1\ell_2...\ell_s$ and $\boldsymbol{\ell}'' = \ell_t\ell_{t+1}...\ell_n$ for some $1 \leq s \leq n,\ 2 \leq t \leq n$ then $\boldsymbol{\ell}' \neq \boldsymbol{\ell}''$ and $\boldsymbol{\ell}'\boldsymbol{\ell}'' \in \mathcal{L}$.*

*Proof.* [Lemma. 5.3.2]

(i) Suppose that $k \geq i$. Then $\boldsymbol{\ell}' = \boldsymbol{vu}$, where $\boldsymbol{v}, \boldsymbol{u}$ are a suffix, prefix of $\boldsymbol{\ell}$ respectively. Using the fact that a Lyndon word precedes any of its suffixes, we have $\boldsymbol{u} < \boldsymbol{\ell} < \boldsymbol{v} < \boldsymbol{vu} < \boldsymbol{u}$, a contradiction.

(ii) Suppose that $\boldsymbol{\ell}' = \boldsymbol{\ell}''$. If $s < t$ then this implies that $\boldsymbol{\ell}$ is bordered contradicting Lyndon properties (Observation 5.3.1). If $s \geq t$ then $\boldsymbol{\ell}', \boldsymbol{\ell}''$ have both prefix and suffix $\ell_t \ell_{t+1} ... \ell_s$, contradicting that the Lyndon words $\boldsymbol{\ell}, \boldsymbol{\ell}', \boldsymbol{\ell}''$ are border-free (Observation 5.3.1). So we have $\boldsymbol{\ell}' \leq \boldsymbol{\ell} < \boldsymbol{\ell}''$, hence $\boldsymbol{\ell}'\boldsymbol{\ell}'' \in \mathcal{L}$ and also $\boldsymbol{\ell}'\boldsymbol{\ell}\boldsymbol{\ell}'', \boldsymbol{\ell}\boldsymbol{\ell}'' \in \mathcal{L}$. □

Note that the above lemma re-expresses known Lyndon properties in our framework, which leads to a partition of the set $F(\mathcal{R}(\boldsymbol{\ell}))$ into those Lyndon factors above or on the diagonal, and those below the diagonal. That is, a Lyndon factor $\boldsymbol{\ell}' = \ell_s \ell_{s+1} ... \ell_t$ is a *prefix factor* if, $\boldsymbol{\ell}' \in \boldsymbol{r}_i$ and the position $j$ of $\ell_s$ is such that $j \geq i$, otherwise $\boldsymbol{\ell}'$ is a *suffix factor*. We denote the set of all prefix factors as *Prefix*, and the set of all suffix factors as *Suffix*. Furthermore, we now show that $\ell_1$ does not occur as a unit factor in *Suffix*, and $\ell_n$ does not occur as a unit factor in *Prefix*, while every factor in *Prefix* starts with $\ell_1$.

**Lemma 5.3.3** *Suppose $\boldsymbol{\ell}' \in F(\mathcal{R}(\boldsymbol{\ell}))$ is a unit factor.*

*(i) If $\boldsymbol{\ell}' \in$ Prefix then $\boldsymbol{\ell}' \neq \ell_n$.*
*(ii) If $\boldsymbol{\ell}' \in$ Suffix then $\boldsymbol{\ell}' \neq \ell_1$.*
*(iii) If $\boldsymbol{\ell}' \in$ Prefix where $\boldsymbol{\ell}' = \ell_s ... \ell_t$ then $\ell_s = \ell_1$.*

*Proof.* [Lemma. 5.3.3]

(i) Suppose that $\boldsymbol{\ell}' = \ell_n$ is a unit factor in row $\boldsymbol{r}_i$ in *Prefix*. Since $n > 1$ and $\ell_1 < \ell_n$ in $\boldsymbol{\ell}$, and since every factor on the diagonal starts $\ell_1$, then there is a first factor $\boldsymbol{w}$ to the left of $\boldsymbol{\ell}'$ in $\boldsymbol{r}_i$ in *Prefix* starting $\ell_1$. If $\ell_n$ belongs to $\boldsymbol{w}$ then $\boldsymbol{\ell}'$ is not a unit factor; otherwise, by Lyndon properties, $\boldsymbol{w}\boldsymbol{\ell}'$ is a Lyndon word but non-unit.

(ii) Suppose that $\boldsymbol{\ell}' = \ell_1$ is a unit factor in *Suffix*. Since $n > 1$ and $\ell_1 < \ell_n$ in $\boldsymbol{\ell}$, then the rightmost factor in row $\boldsymbol{r}_i$ in *Suffix* ends $\ell_n$. Let $\boldsymbol{w}$ be the first factor right of $\boldsymbol{\ell}'$ in row $\boldsymbol{r}_i$ in *Suffix* ending $\ell_n$. Either $\boldsymbol{w}$ starts with $\ell_1$, or $\ell_1\boldsymbol{w}$ is a Lyndon factor (which does not cross the diagonal), contradicting $\boldsymbol{\ell}'$ is a unit factor.

(iii) Let the factorization of row $\boldsymbol{r}_i$ in *Prefix* be $\boldsymbol{\ell}_1 \geq \boldsymbol{\ell}_2 \geq ... \geq \boldsymbol{\ell}_k$, then $\boldsymbol{\ell}_1$ commences with $\ell_1$. Since $\boldsymbol{\ell}$ is a Lyndon word then there is no letter less than $\ell_1$ in the factorization. So suppose that $\boldsymbol{\ell}_2$ commences with a letter $\ell_2 > \ell_1$, then $\boldsymbol{\ell}_1\boldsymbol{\ell}_2$

is a Lyndon word contradicting the factorization. □

Similarly we see that a maximal letter $\ell_s$ cannot occur in *Prefix* as a unit factor; a unit factor in *Prefix* can only be $\ell_1$. Further, from Lemma 5.3.3 (iii) above, we can deduce that a factorization $\boldsymbol{\ell}_p \geq \boldsymbol{\ell}_{p+1} \geq \cdots \geq \boldsymbol{\ell}_m$ of a row in *Prefix* is in a sense 'periodic' with period $\boldsymbol{\ell}_p$.

**Lemma 5.3.4** *Suppose the factorization of row $\boldsymbol{r}_i$ is $\boldsymbol{\ell}_1 \geq \boldsymbol{\ell}_2 \geq \cdots \geq \boldsymbol{\ell}_p \geq \cdots \geq \boldsymbol{\ell}_{p'}$ where $\boldsymbol{\ell}_p$ is the diagonal factor. Then $\boldsymbol{\ell}_{p+t} = \boldsymbol{\ell}_p$ for $1 \leq t < p' - p$ and $\boldsymbol{\ell}_{p'}$ is a prefix of $\boldsymbol{\ell}_p$.*

*Proof.* [Lemma. 5.3.4]

Since $\boldsymbol{\ell}_p$ is the diagonal factor, we have $\boldsymbol{\ell}_p \in$ *Prefix* and $\boldsymbol{\ell}_{p-1} \notin$ *Prefix*. By the factorization we have $\boldsymbol{\ell}_p \geq \boldsymbol{\ell}_{p+t}$, so suppose that $\boldsymbol{\ell}_p > \boldsymbol{\ell}_{p+t}$. If $|\ \boldsymbol{\ell}_p\ | = |\ \boldsymbol{\ell}_{p+t}\ |$, then since the given Lyndon word $\boldsymbol{\ell}$ commences with $\boldsymbol{\ell}_p$, this contradicts that $\boldsymbol{\ell}$ is a Lyndon word. If $|\ \boldsymbol{\ell}_p\ | > |\ \boldsymbol{\ell}_{p+t}\ |$ then similarly $\boldsymbol{\ell}$ cannot be a Lyndon word. Finally suppose that $|\ \boldsymbol{\ell}_p\ | < |\ \boldsymbol{\ell}_{p+t}\ |$. If $\boldsymbol{\ell}_p$ is a proper prefix of $\boldsymbol{\ell}_{p+t}$ then this contradicts $\boldsymbol{\ell}_p > \boldsymbol{\ell}_{p+t}$, so some $i$th letter of $\boldsymbol{\ell}_p$ (with minimal $i$) must be greater than the $i$th letter of $\boldsymbol{\ell}_{p+t}$ contradicting that $\boldsymbol{\ell}$ a Lyndon word. Hence $\boldsymbol{\ell}_p = \boldsymbol{\ell}_{p+t}$. Note that $\boldsymbol{\ell}_{p'}$ is not necessarily a proper prefix of $\boldsymbol{\ell}_p$. If we suppose that $|\ \boldsymbol{\ell}_{p'}\ | \geq |\ \boldsymbol{\ell}_p\ |$, then the above argument shows that $\boldsymbol{\ell}_{p'} = \boldsymbol{\ell}_p$. Otherwise, if $|\ \boldsymbol{\ell}_{p'}\ | < |\ \boldsymbol{\ell}_p\ |$ then by the factorization $\boldsymbol{\ell}_p > \boldsymbol{\ell}_{p'}$, and since $\boldsymbol{\ell}$ is a Lyndon word then there is no $i$th letter in $\boldsymbol{\ell}_{p'}$ which is less than the $i$th letter in $\boldsymbol{\ell}_p$. Hence all corresponding $i$th letters are equal, and so $\boldsymbol{\ell}_{p'}$ is a proper prefix of $\boldsymbol{\ell}_p$. □

Defining the sets *Prefix* and *Suffix* leads to an extension of Lemma 5.3.2.

**Lemma 5.3.5** *Let $\boldsymbol{\ell} = \ell_1\ell_2...\ell_n \in \mathcal{L}$, and let $\boldsymbol{\ell}', \boldsymbol{\ell}'' \in F(\mathcal{R}(\boldsymbol{\ell}))$. If $\boldsymbol{\ell}' \in$ Prefix and $\boldsymbol{\ell}'' \in$ Suffix then $\boldsymbol{\ell}' \neq \boldsymbol{\ell}''$ and $\boldsymbol{\ell}' < \boldsymbol{\ell}''$.*

*Proof.* [Lemma. 5.3.5]

If $\boldsymbol{\ell}' = \boldsymbol{\ell}$, then since $|\ \boldsymbol{\ell}'\ | > |\ \boldsymbol{\ell}''\ |$ then clearly $\boldsymbol{\ell}' \neq \boldsymbol{\ell}''$. So suppose that $\boldsymbol{\ell}' = \boldsymbol{\ell}''$. By Lemma 5.3.4, $\boldsymbol{\ell}'$ is a proper prefix of $\boldsymbol{\ell}$. Let the factorization containing $\boldsymbol{\ell}''$ in *Suffix* be $\boldsymbol{\ell}_1 \geq \boldsymbol{\ell}_2 \geq ... \geq \boldsymbol{\ell}_k$, then $\boldsymbol{\ell}'' \geq \boldsymbol{\ell}_k$. Since $\boldsymbol{\ell}_k$ is a (proper) suffix of $\boldsymbol{\ell}$, then by Lyndon properties we have $\boldsymbol{\ell}' < \boldsymbol{\ell}_k \leq \boldsymbol{\ell}''$ and we conclude that $\boldsymbol{\ell}'$ and $\boldsymbol{\ell}''$ are distinct

with $\boldsymbol{\ell}' < \boldsymbol{\ell}''$. □

It follows that if $\mathit{Prefix}_{lex}$, $\mathit{Suffix}_{lex}$ are the sets *Prefix* and *Suffix* in lexicographic order respectively, then
$\mathit{Prefix}_{lex} < \mathit{Suffix}_{lex}$. We now relate a well-known property, that any Lyndon word can be split into two ordered Lyndon words, with the number of factors in $F(\mathcal{R}(\ell))$.

**Lemma 5.3.6** *Let $\boldsymbol{\ell} \in \mathcal{L}$, and let $k \geq 1$ be the number of ways of splitting $\boldsymbol{\ell}$ into two Lyndon words. Then in $F(\mathcal{R}(\boldsymbol{\ell}))$*

*(i) there is exactly one row with only one factor,*
*(ii) there are exactly $k$ rows with only two factors.*

*Proof.* [Lemma. 5.3.6]

(i) By definition of the uniqueness of a Lyndon word, the first row (only) of $F(\mathcal{R}(\boldsymbol{\ell}))$ is Lyndon and hence consists of exactly one factor.
(ii) Let $\boldsymbol{\ell} = \boldsymbol{uv}$, then the $\mid v \mid$th rotation, giving the $\mid v \mid +1$th row, consists of the two factors $\boldsymbol{v} > \boldsymbol{u}$, where $\boldsymbol{v} \in$ *Suffix* and $\boldsymbol{u} \in$ *Prefix*. This holds for each of the $k$ distinct splits of $\boldsymbol{\ell}$. □

For $\boldsymbol{\ell} = \ell_1\ell_2...\ell_n$, from Lyndon properties we have $\ell_1 < \ell_n$, and so we consider the occurrences of $\ell_1, \ell_n$ in $F(\mathcal{R}(\boldsymbol{\ell}))$.

**Lemma 5.3.7** *Let $j, k$ be the frequency of $\ell_1, \ell_n$ in $\boldsymbol{\ell}$ respectively. Then $j$ rows end with the unit factor $\ell_1$, and if $\ell_n$ is maximal in $\boldsymbol{\ell}$ then $k$ rows start with the unit factor $\ell_n$.*

*Proof.* [Lemma. 5.3.7]

Consider the $j$ rotations which end $\ell_1$, hence $\ell_1$ is in position $n$ in each of these rows. Since $\ell_1$ is minimal in $\boldsymbol{\ell}$, then the Lyndon factor to its left starts with a letter greater than or equal to $\ell_1$, and so we cannot extend the last letter $\ell_1$ leftwards. Hence, $\ell_1$ must occur at the end of a row as a unit factor, with the last occurrence being in row $n$. Similarly consider the $k$ rotations which commence with $\ell_n$. If $\ell_n$ is maximal in

$\boldsymbol{\ell}$, then if the Lyndon factor to its right starts with a letter less than $\ell_n$ then we cannot concatenate $\ell_n$ with this factor; if the factor to the right is $\ell_n$ again we cannot concatenate as it would form a repetition. Hence each of the $k$ occurrences of $\ell_n$ at the start of rows comprise a unit factor, with the first occurrence being in row 2. Similarly, if $\ell_s \neq \ell_n$ is maximal in $\boldsymbol{\ell}$ and occurs with frequency $i$, then $i$ rows start with the unit factor $\ell_s$, while the second row still commences with the unit factor $\ell_n$ followed by a factor starting with $\ell_1$. □

The following lemma shows that we can start constructing $F(\mathcal{R}(\boldsymbol{\ell}))$ iteratively from Lyndon sub-words (substrings):

**Lemma 5.3.8** *Let $\boldsymbol{\ell}, \boldsymbol{\ell}' \in \mathcal{L}$ with $\boldsymbol{\ell} < \boldsymbol{\ell}'$.Then*

*(i)* *Suffix*$(F(\mathcal{R}(\boldsymbol{\ell}'))) \subset$ *Suffix*$(F(\mathcal{R}(\boldsymbol{\ell}\boldsymbol{\ell}')))$,
*(ii)* *Prefix*$(F(\mathcal{R}(\boldsymbol{\ell}))) \subset$ *Prefix*$(F(\mathcal{R}(\boldsymbol{\ell}\boldsymbol{\ell}')))$.

*Proof.* [Lemma. 5.3.8]

From Lemma 5.3.2 we know that no Lyndon word crosses the diagonal, so we can construct the *Prefix* and *Suffix* sets independently. Hence the rows, $r_2$ to $r_{|\boldsymbol{\ell}'|}$ in *Suffix*$(F(\mathcal{R}(\boldsymbol{\ell}\boldsymbol{\ell}')))$, are given by the corresponding rows in *Suffix*$(F(\mathcal{R}(\boldsymbol{\ell}')))$, while $r_{|\boldsymbol{\ell}'|+1}$ is $\boldsymbol{\ell}'$. Similarly the rows, $r_{n-|\boldsymbol{\ell}|+1}$ to $r_n$ in *Prefix*$(F(\mathcal{R}((\boldsymbol{\ell}\boldsymbol{\ell}'))$, are given by the rows $r_1$ to $r_{|\boldsymbol{\ell}|}$ in *Prefix*$(F(\mathcal{R}(\boldsymbol{\ell}))$. □

It follows that for Lyndons $\boldsymbol{\ell} < \boldsymbol{\ell}'$, if $\boldsymbol{\ell}_{lex}$, $\boldsymbol{\ell}'_{lex}$ is the lexicographic order of all the factors in $F(\mathcal{R}(\boldsymbol{\ell}))$, $F(\mathcal{R}(\boldsymbol{\ell}'))$ respectively, then $\boldsymbol{\ell}_{lex} < \boldsymbol{\ell}'_{lex}$. Consider the behaviour of a Lyndon factor as it moves down $F(\mathcal{R}(\boldsymbol{\ell}))$ one row at a time. If it belongs to *Suffix* it may get bigger but never smaller, hence it is largest in the bottom row $r_n$. Whereas if it belongs to *Prefix*, then it never gets bigger but may itself be factored. So we can think of *Prefix* as the *factorization* set, and *Suffix* as the *concatenation* set. We further see that factors form *diagonal strips* in $F(\mathcal{R}(\boldsymbol{\ell}))$, that eventually get narrower in *Prefix* but wider in *Suffix*. Clearly if $\boldsymbol{\ell} = \lambda_1 ... \lambda_j$ is a Lyndon word, then $\boldsymbol{\ell}' = \lambda_1 ... \lambda_{j-1}$ may either also be Lyndon, or it factors as $(\boldsymbol{\ell}'_1)(\boldsymbol{\ell}'_2)...(\boldsymbol{\ell}'_t)$. However, the next lemma shows that if the Lyndon factors $\boldsymbol{\ell}, \boldsymbol{\ell}'$ are in row $r_i$ in *Prefix* with $\boldsymbol{\ell} \geq \boldsymbol{\ell}'$, and if the last letter in $\boldsymbol{\ell}'$ is rotated reducing $\boldsymbol{\ell}'$ to $\boldsymbol{\ell}''$, then the factor $\boldsymbol{\ell}$ is unchanged in row $r_{i+1}$.

**Lemma 5.3.9** *Let $\ell, \ell' \in \mathcal{L}$ where $\ell = \lambda_1 ... \lambda_j$ and $\ell' = \mu_1 ... \mu_k$ with $\ell \geq \ell'$. Let $\ell'' = \mu_1 ... \mu_{k-1}$ and $f(\ell'') = (\ell''_1)(\ell''_2)...(\ell''_t)$ then $\ell > \ell'' \geq \ell''_1$.*

*Proof.* [Lemma. 5.3.9]

If $\ell'$ is equal to $\ell$ or a proper prefix of $\ell$, then $\ell \geq \ell' > \ell'' \geq \ell''_1$ (note $\ell''$ may be $\varepsilon$). Otherwise, let $i$ be minimal such that $\lambda_i \neq \mu_i$, hence $\lambda_i > \mu_i$. If $i = k$, then $\ell''$ is a proper prefix of $\ell$ ($\ell''$ may be $\varepsilon$) and so $\ell > \ell'' \geq \ell''_1$. Otherwise, if $i < k$ then $\ell > \ell' > \ell'' \geq \ell''_1$. □

We now show that the Lyndon factors over consecutive rows in *Suffix* can become concatenated as the next letter $\lambda$ is rotated around, so that the number $k$ of factors in *Suffix* in row $r_n$ is $1 \leq k \leq n-1$; since no factor crosses the *diagonal*, the total number of factors $k'$ in both *Suffix* and *Prefix* in row $r_n$ is $2 \leq k' \leq n$.

**Lemma 5.3.10** *Let $\ell_1, \ell_2, ..., \ell_k \in \mathcal{L}$ with $\ell_1 \geq \ell_2 \geq ... \geq \ell_k$. If $\ell_1$ is the first letter of $\ell_1$, $s$ is the index such that $\lambda\ell_1...\ell_s \in \mathcal{L}$ and $\lambda \in \Sigma$, then*

*(i) if $\lambda \leq \ell_1$ then $f(\lambda\ell_1\ell_2...\ell_k) = (\lambda\ell_1...\ell_s)(\ell_{s+1})...(\ell_t)$ where $1 \leq t \leq k$,*
*(ii) if $\lambda > \ell_1$ then $f(\lambda\ell_1\ell_2...\ell_k) = \lambda > \ell_1 \geq \ell_2 ... \geq \ell_k$.*

*Proof.* [Lemma. 5.3.10]

(i) If $\lambda < \ell_1$ then $\lambda\ell_1 \in \mathcal{L}$. Let $s$ be the index such that $\lambda\ell_1...\ell_s \in \mathcal{L}$, and $\lambda\ell_1...\ell_s \geq \ell_{s+1}$, that is at least the first factor is extended. If $\lambda = \ell_1$, hence $\ell_1$ is a unit factor, then $f(\lambda\ell_1\ell_2...\ell_k) = \lambda \geq \ell_1 \geq \ell_2 \geq ... \geq \ell_k$. Note that from concatenation properties of Lyndon words, $\lambda\ell_1...\ell_s$ does not "overlap" any of the prefixes of the factors $\ell_1, \ell_2, ..., \ell_k$, that is, entire factors may be concatenated but not split.
(ii) This is obvious, and no factors are concatenated. □

Using the preliminary results above, we can introduce the *Lyndon fountain*. Let $D$ be the diagonal in reverse sequence, that is given $F(\mathcal{R}(\ell))$, $D$ is the sequence of Lyndon factors $\ell_n...\ell_1$, such that $\ell_i$ starts at position $i$ in $f(\boldsymbol{r_i})$. Also, if $k$ is the number of unit factors $\ell_1$ in *Prefix*, let $P'$ denote the *Prefix* set minus all of the $k$ factors $\ell_1$.

**Lemma 5.3.11** *Given $F(\mathcal{R}(\boldsymbol{\ell}))$, let $\pi$ be any permutation (possibly empty) of the factors in the Suffix set. Then*

*(i)* $\boldsymbol{\ell}\pi \in \mathcal{L}$.
*(ii)* $D\boldsymbol{\ell}^j\pi \in \mathcal{L}$ *for* $j \geq 0$.
*(iii)* $\ell_1^k P'\boldsymbol{\ell}\pi \in \mathcal{L}$.

*Proof.* [Lemma. 5.3.11]

(i) This follows from the Lyndon property that a Lyndon word precedes any of its proper suffixes in lexicographic order. Suppose $\boldsymbol{\ell}' \in$ *Suffix* is adjacent to the diagonal in row $r_i$, then $\boldsymbol{\ell}'$ is a proper suffix of $\boldsymbol{\ell}$ and so $\boldsymbol{\ell}\boldsymbol{\ell}' \in \mathcal{L}$. Suppose $\boldsymbol{\ell}'' \in$ *Suffix* is also in row $r_i$ and does not end in position $i-1$, that is, it is not adjacent to the diagonal. Then by the factorization of row $r_i$, we have $\boldsymbol{\ell}'' \geq \boldsymbol{\ell}'$, hence $\boldsymbol{\ell}\boldsymbol{\ell}'', \boldsymbol{\ell}\boldsymbol{\ell}''\boldsymbol{\ell}', \boldsymbol{\ell}\boldsymbol{\ell}'\boldsymbol{\ell}'' \in \mathcal{L}$. This argument extends naturally to any permutation of factors, including those from distinct rows, in *Suffix*. (Note that these concatenations do not form a repetition.)

(ii) The factors $\boldsymbol{\ell}_n ... \boldsymbol{\ell}_2$ in $D$ (which all begin $\ell_1$) are all proper prefixes of $\boldsymbol{\ell}$, starting with $\boldsymbol{\ell}_n = \ell_1$, and $D$ ends with $\boldsymbol{\ell}_1 = \boldsymbol{\ell}$; further, applying Lemma 5.3.9, $\boldsymbol{\ell}_i$ is either equal to or a proper prefix of $\boldsymbol{\ell}_{i-1}$, giving $\boldsymbol{\ell}_n \leq \boldsymbol{\ell}_{n-1} \leq \boldsymbol{\ell}_{n-2} \leq ... \leq \boldsymbol{\ell}_2 < \boldsymbol{\ell}_1 = \boldsymbol{\ell}$. Hence, the factors in $D$ are in lexicographic order, which also do not form a repetition. The rest follows from part (i).

(iii) Clearly $\ell_1^k$ is a maximal run of minimal letters (note that $k$ is greater than or equal to the number of occurrences of $\ell_1$ in $\boldsymbol{\ell}$). We then concatenate any permutation of remaining prefix factors followed by (any repetitions of) $\boldsymbol{\ell}$ followed by suffix factors. Again due to the run of minimal letters at the start then this does not generate a repetition. □

So we see that starting with the word $\boldsymbol{\ell}$, the factorization set of this Lyndon word behaves like a *fountain* yielding many more Lyndon words over the restricted alphabet $\Sigma$ given by $\boldsymbol{\ell}$; similarly we also get the Lyndon words $\ell_1^k D$, $\ell_1^k \pi$ (if $\pi$ is non-empty), basically any permutation of any factors after the run $\ell_1^k$, and many more by taking factors from subsets of the sets *Prefix* and *Suffix*.

We conclude by outlining two related algorithms. First consider computing the set $F(\mathcal{R}(\boldsymbol{\ell}))$. For this we can use Duval's linear Lyndon factorization algorithm [Duv83]: a naive application is $\mathcal{O}(n^2)$; we explain a more efficient implementation where computing *Prefix* is linear while *Suffix* is quadratic. When scanning the input Lyndon word $\boldsymbol{\ell} = \ell_1\ell_2...\ell_n$ from left to right, intermediate Lyndon factors are detected (each beginning with $\ell_1$). As each factor is found, then the set *Prefix* can be constructed from row $r_n$ to row $r_1$, where of course $r_1$ is the given word $\boldsymbol{\ell}$. The set *Suffix* is also constructed from row $r_n$ to row $r_2$, starting with $r_n = f(\ell_2...\ell_n) = (\boldsymbol{x_1})(\boldsymbol{x_2})...(\boldsymbol{x_m})$. As we work up the rows we need only re-factor the current leftmost word as it is repeatedly being decremented; the other factors are unchanged (by the monotonicity of factorizations and a Lyndon word precedes its suffixes). Let $\boldsymbol{x_1} = x_{1_1}x_{1_2}...x_{1_j}$, then row $r_{n-1}$ is given by $f(x_{1_2}...x_{1_j})(\boldsymbol{x_2})...(\boldsymbol{x_m})$ (with complexity $\mathcal{O}(n)$), and we continue re-factoring the shrinking $\boldsymbol{x_1}$ until row $r_{n-|\boldsymbol{x_1}|} = (\boldsymbol{x_2})...(\boldsymbol{x_m})$; similarly proceed up to row $r_2$ to complete the factorization of *Suffix*.

Next we show how to obtain the BWT of a Lyndon substring $\boldsymbol{x}$ of $\boldsymbol{\ell}$. Start by identifying $\boldsymbol{x} = x_i...x_j$ and its starting and ending positions $\boldsymbol{x}[i]$, $\boldsymbol{x}[j]$ in $\boldsymbol{\ell}$ using a linear scan. Similarly associate each letter in $\boldsymbol{x}$ with its position. The first letter in BWT($\boldsymbol{x}$) is $x_j$. Subsequently consider those rows of $F(\mathcal{R}(\boldsymbol{\ell}))$ which start with a suffix of $\boldsymbol{x}$ and hence end with a prefix of $\boldsymbol{x}$. That is, for each row in $\mathcal{R}(\boldsymbol{\ell})^<$ starting with a letter which is in position $\boldsymbol{x}[i] < k \leq \boldsymbol{x}[j]$, then taking those rows in their order of occurrence, the BWT($\boldsymbol{x}[k]$) is the rightmost letter of that row. Since the relevant rows are in lexicographic order in $\mathcal{R}(\boldsymbol{\ell})^<$, then this gives the required order for the letters in BWT($\boldsymbol{x}$). Note that the rows can be sorted in linear time due to a clever suffix array technique [KA03].

Also note that not every Lyndon substring of a Lyndon word necessarily occurs in $F(\mathcal{R}(\boldsymbol{\ell}))$, for example, if $\boldsymbol{\ell} = abccad$ then the factor $bc$ does not occur in $F(\mathcal{R}(abccad))$. Some future lines of inquiry are: to characterize the factors that do occur in $F(\mathcal{R}(\boldsymbol{\ell}))$; to apply the structure $F(\mathcal{R}(\boldsymbol{\ell}))$ for efficiently computing the BWT; and to consider the structures presented here for Lyndon subsequences of strings.

# Article: # 6

# Specialized Border and Suffix Arrays

We consider the problem of finding repetitive structures and inherent patterns in a given string $s$ over a finite totally ordered alphabet. We combine the well-known concepts of Lyndon words, borders and suffix arrays to introduce the Lyndon Border Array and the Lyndon Suffix Array. We present linear time and space algorithms for computing these two interesting data structures.

---

## 6.1 Introduction

Understanding complex patterns and repetitive structures in strings is essential for efficiently solving many problems in stringology [CR02]. For instance, Lyndon words are increasingly a fundamental and applicable form in the study of combinatorics on words [Lot83], [Lot05], [Smy03] - these patterned words have deep links with algebra and are rich in structural properties. Another important concept is a border $u$ of a string $s$ defined to be both a prefix and a suffix of $s$ such that $u \neq s$. The computation of the border array of a string $s$, that is of the borders of each prefix of $s$, is strongly related to the string matching problem: given a string $w$, find all of its occurrences in a string $s$. It constitutes the "failure function" of the Morris-Pratt (1970) string matching algorithm [MP70].

Lyndon words were introduced under the name of ***standard lexicographic sequences*** [Lyn54, Lyn55] in order to construct a basis of a free abelian group. Two strings are ***conjugate*** if they differ only by a cyclic permutation of their characters; a *Lyndon word* is defined as a (generally) finite word which is strictly minimal for the lexicographic order of its conjugacy class. For a non-letter Lyndon word $w$, the pair $(u, v)$ of Lyndon words such that $w = uv$ with $v$ of maximal length is called the ***standard factorization*** of $w$.

The set of Lyndon words permits the unique maximal factorization of any given string [CFL58, Lot83]. In 1983, Duval [Duv83] developed an algorithm for standard factorization that runs in linear time and space – the algorithm cleverly iterates over a string trying to find the longest Lyndon word; when it finds one, it adds it to the result list and proceeds to search in the remaining part of the string.

Lyndon words proved to be useful for constructing bases in free Lie algebras [Reu93], constructing de Bruijn sequences [FM78], computing the lexicographically smallest or largest substring in a string [AC95], succinct suffix-prefix matching of highly periodic strings [NS13]. Wider ranging applications include the Burrows-Wheeler transform and data compression [GS12], musicology [Che04], bioinformatics [DR04], and in relation to cryptanalysis [Per05]. Indeed the uses, and hence importance, of Lyndon words are increasing, and so we are motivated to investigate specialized Lyndon data structures.

The key contributions of this article are as follows.

- By combining the important concepts of Lyndon words and borders of strings, we introduce here the *Lyndon Border Array* $\mathcal{L}\beta$ of $\boldsymbol{s}$, whose $i$-th entry $\mathcal{L}\beta(\boldsymbol{s})[i]$ is the length of the longest border of $\boldsymbol{s}[1\,..\,i]$ which is also a Lyndon word. We present an efficient linear time and space algorithm for computing the *Lyndon Border Array* $\mathcal{L}\beta$ for a given string (Section 6.4).

- In order to achieve the desired level of efficiency in the Lyndon Border Array construction we also present some interesting results related to Lyndon combinatorics, which we believe is of independent interest as well (Section 6.3).

- A complementary data structure, the *Lyndon Suffix Array*, which is an adaptation of the classic suffix array, is also defined; by modifying the linear-time construction of Ko and Aluru [KA03] we similarly achieve a linear construction for our Lyndon variant (Section 6.5). We also present a simpler algorithm to construct a Lyndon Suffix Array from a given Suffix Array (Section 6.5.1). The latter algorithm also runs in linear time and space.

## 6.2 Basic Definitions and Notations

Consider a finite totally ordered alphabet $\Sigma$ which consists of a set of characters (equivalently letters or symbols). The cardinality of the alphabet is denoted by $|\Sigma|$.

For a substring $\boldsymbol{w}$ of $\boldsymbol{s}$, the string $\boldsymbol{uwv}$ for $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^*$ is an extension of $\boldsymbol{w}$ in $\boldsymbol{s}$ if $\boldsymbol{uwv}$ is a substring of $\boldsymbol{s}$; $\boldsymbol{wv}$ for $\boldsymbol{v} \in \Sigma^*$ is the right extension of $\boldsymbol{w}$ in $\boldsymbol{s}$ if $\boldsymbol{wv}$ is a substring of $\boldsymbol{s}$; $\boldsymbol{uw}$ for $\boldsymbol{u} \in \Sigma^*$ is a left extension of $\boldsymbol{w}$ in $\boldsymbol{s}$ if $\boldsymbol{uw}$ is a substring of $\boldsymbol{s}$. Words that are both prefixes and suffixes of $\boldsymbol{w}$ are called borders of $\boldsymbol{w}$. By $border(\boldsymbol{w})$ we denote the length of the longest border of $\boldsymbol{w}$ that is shorter than $\boldsymbol{w}$.

A word $\boldsymbol{w}$ is ***periodic*** if it can be expressed as $\boldsymbol{w} = \boldsymbol{p}^k\boldsymbol{p}'$ where $\boldsymbol{p}'$ is a proper prefix of $p$, and $k \geq 2$. Moreover, a string is said to be ***primitive*** if it cannot be written as $\boldsymbol{u}^k$ with $\boldsymbol{u} \in \Sigma^+$ and $k \geq 2$, i.e., it is not a power of another string. When $\boldsymbol{p}$ is primitive, we call it "the period" of $\boldsymbol{u}$. It is a known fact [CHL07] that, for any string $\boldsymbol{w}$, $per(\boldsymbol{w}) + border(\boldsymbol{w}) = |\boldsymbol{w}|$, where the period ***per*** of a nonempty string is the smallest of its periods.

**Definition 29** *(Border array) For a string* $\boldsymbol{s} \in \Sigma^n$*, the border array* $\beta(\boldsymbol{s})[1\,..\,n]$ *is defined by* $\beta(\boldsymbol{s})[i] = |border(\boldsymbol{s}[1\,..\,i])|$ *for* $1 \leq i \leq n$*.*

**Proposition 6.2.1** *[MP70] The border of a string* $\boldsymbol{s}$ *(or the table* $\beta(\boldsymbol{s})$ *itself) can be computed in time* $\mathcal{O}(|\boldsymbol{s}|)$*.*

A string $\boldsymbol{y} = \boldsymbol{y}[1\,..\,n]$ is a ***conjugate*** (or cyclic rotation) of $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$ if $\boldsymbol{y}[1\,..\,n] = \boldsymbol{x}[i\,..\,n]\boldsymbol{x}[1\,..\,i-1]$ for some $1 \leq i \leq n$ (for $i = 1$, $\boldsymbol{y} = \boldsymbol{x}$). A ***Lyndon word*** is a primitive word which is minimal for the lexicographical order of its conjugacy class (i.e., the set of all words obtained by cyclic rotations of letters). Furthermore, a non-empty word is a Lyndon word if and only if it is strictly smaller in lexicographical order (lexorder) than any of its non-empty proper suffixes [Duv83, Lot83].

Throughout this article, $\mathcal{L}$ will denote the set of Lyndon words over the totaly ordered alphabet $\Sigma$, and $\mathcal{L}_n$ will denote the set of Lyndon words of length $n$; hence $\mathcal{L} = \{\mathcal{L}_1 \cup \mathcal{L}_2 \cup \mathcal{L}_3 \ldots\}$. We next list several well-known properties of Lyndon words and border arrays which we later apply to develop the new algorithms.

**Proposition 6.2.2** *[Duv83] A word* $\boldsymbol{w} \in \Sigma^+$ *is a Lyndon word if and only if either* $\boldsymbol{w} \in \Sigma$ *or* $\boldsymbol{w} = \boldsymbol{uv}$ *with* $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{L}$*,* $\boldsymbol{u} < \boldsymbol{v}$*.*

**Theorem 6.2.3** *[CFL58] Any word* $\boldsymbol{w}$ *can be written uniquely as a non-increasing product* $\boldsymbol{w} = \boldsymbol{u}_1\boldsymbol{u}_2 \cdots \boldsymbol{u}_k$ *of Lyndon words.*

Theorem 6.2.3 shows that there is a unique decomposition of any word into non-increasing Lyndon words ($\boldsymbol{u}_1 \geq \boldsymbol{u}_2 \geq \cdots \geq \boldsymbol{u}_k$).

**Observation 6.2.4** *Let* $\boldsymbol{\ell}$ *be a Lyndon word (*$\boldsymbol{\ell} \in \mathcal{L}$*) where* $\boldsymbol{\ell} = \ell_1\ell_2\,..\,\ell_n$ *(to avoid trivialities we assume* $n > 1$*), then*

*(1)* $border(\boldsymbol{\ell}) = 0$*,*

*(2)* $\ell_1 < \ell_n$*,*

*(3)* $\ell_1 \leq \ell_i | \ell_i \in \{\ell_2, \ldots, \ell_{n-1}\}$*,*

*(4)* $\boldsymbol{\ell} < \ell_i \cdots \ell_n$*, for* $1 < i \leq n$*.*

**Observation 6.2.5** *Given a string $\boldsymbol{s}$, then*

*(1)* $\beta(\boldsymbol{s})[1] = 0$,

*(2) if $\mathbf{b}$ is a border of $\boldsymbol{s}$, and $\mathbf{b}'$ is a border of $\mathbf{b}$, then $\mathbf{b}'$ is a border of $\boldsymbol{s}$,*

*(3)* $0 \leq \beta(\boldsymbol{s})[i+1] \leq \beta(\boldsymbol{s})[i] + 1$, *for* $1 \leq i < n$.

We now introduce the *Lyndon Border Array* and associated computation, illustrated in Example 6.2.6 below.

**Definition 30** *(Lyndon Border Array) For a string $\boldsymbol{s} \in \Sigma^n$, the* Lyndon border array *$\mathcal{L}\beta(\boldsymbol{s})[i]$ is the length of the longest border of $\boldsymbol{s}[1\,..\,i]$ which is also a Lyndon word.*

**Definition 31 (Lyndon suffix array)** *For a string $\boldsymbol{s} \in \Sigma^n$, the* Lyndon Suffix Array *of $\boldsymbol{s}$ is the lexicographically sorted list of all those suffixes of $\boldsymbol{s}$ that form Lyndon words.*

Given a string $\boldsymbol{s}$ of length $n$, associated computational problems are: compute the Lyndon border and Lyndon suffix arrays; we address these problems in this article.

**Example 6.2.6** *Consider the string $\boldsymbol{s} = abaabaaabbaabaab$. The following table illustrate the border array $\beta$ of $\boldsymbol{s}$, the Lyndon border array $\mathcal{L}\beta$ of $\boldsymbol{s}$, the suffix array $\mathcal{A}$ of $\boldsymbol{s}$ and the Lyndon suffix array $\mathcal{L}S$ of $\boldsymbol{s}$.*

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{s}[i]$ | a | b | a | a | b | a | a | a | b | b | a | a | b | a | a | b |
| $\beta[i]$ | 0 | 0 | 1 | 1 | 2 | 3 | 4 | 1 | 2 | 0 | 1 | 1 | 2 | 3 | 4 | 5 |
| $\mathcal{L}\beta[i]$ | 0 | 0 | 1 | 1 | 2 | 1 | 1 | 1 | 2 | 0 | 1 | 1 | 2 | 1 | 1 | 2 |
| $\mathcal{A}[i]$ | 5 | 13 | 2 | 10 | 6 | 14 | 3 | 11 | 0 | 7 | 15 | 4 | 12 | 1 | 9 | 8 |
| $\mathcal{L}S[i]$ | 5 | 13 | 14 | 15 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

## 6.3 Lyndon Combinatorics

This section introduces some new interesting combinatorial results on Lyndon words. In relation to the computation of the Lyndon Border Array, we here show how to find the shortest prefix of a string that is both border-free and not a Lyndon word. So assume that for a given string $\boldsymbol{s}$ of length $n$, we have $\boldsymbol{s}[1] = \gamma$. If $\boldsymbol{f_1}, \ldots, \boldsymbol{f_q}$ are factors of

$\boldsymbol{s}$, we use $start(\boldsymbol{f_i})$ $(end(\boldsymbol{f_i}))$ to denote the index of $\boldsymbol{f_i}[1]$ $(\boldsymbol{f_i}[|\boldsymbol{f_i}|])$ in $\boldsymbol{s}$, and say that $j$ is an index of $\boldsymbol{f_i}$ if $start(\boldsymbol{f_i}) \leq j \leq end(\boldsymbol{f_i})$. An outline of the steps of the algorithm is as follows:

Algorithm *Shortest non-Lyndon Border-free Prefix (SNLBfP).*

1. Compute the Lyndon factorization of $\boldsymbol{s}$.
2. Apply binary search to find the first Lyndon factor $\boldsymbol{f_\mu}$ in the factorization starting with the largest letter $\mu$ which is strictly less than $\gamma$ (if it exists).
3. Consider the maximal prefix $\boldsymbol{p}$ of $\boldsymbol{s}$ in which every factor $\boldsymbol{f_1}, \ldots, \boldsymbol{f_q}$ starts with $\gamma$; compute the border array $\beta(\boldsymbol{p})$ of $\boldsymbol{p}$.
4. Compute $i$, the smallest index of $\boldsymbol{p}$, such that $i > end(\boldsymbol{f_1})$ ($i$ is not an index of $\boldsymbol{f_1}$) and $\beta(\boldsymbol{p})[i] = 0$;
   (a) if $i$ does not exist then $i = end(\boldsymbol{f_q}) + 1$ (if $\boldsymbol{p}$ exists)
   (b) if $q = 1$ then $i = end(\boldsymbol{f_1}) + 1$ (if $\boldsymbol{p}$ exists).
5. Return $\boldsymbol{s}[1 \ldots i]$.

**Claim 6.3.1** *Suppose $\boldsymbol{s}[1 \ldots i]$ is the shortest prefix of $\boldsymbol{s}$ that is both border-free and non-Lyndon. Then $i > end(\boldsymbol{f_1})$, i.e., Algorithm SNLBfP is correct in skipping the first Lyndon factor.*

*Proof.* [Proof of Claim 6.3.1] The proof is by induction. Assume that the first Lyndon factor $\boldsymbol{f_1}$ is of length $m$ (with $m \geq 2$ otherwise the Claim holds trivially). By Observation 6.2.4((1)) and Observation 6.2.5((1)) we have $\beta(\boldsymbol{f_1})[1] = \beta(\boldsymbol{f_1})[m] = 0$. The smallest $j$ after 1 where $\beta(\boldsymbol{f_1})[j] = 0$ must index a Lyndon word $\boldsymbol{x}$ of the form $\boldsymbol{x} = \gamma^{j-1}\nu$, where $\nu > \gamma$. Hence, after the first letter (which is a Lyndon word), the next border-free prefix is also a Lyndon word.

Assume now that all border-free prefixes up to index $t$ of $\boldsymbol{f_1}$ are Lyndon words (and hence nested), and suppose that the next border-free position is $t'$. Then we need to show that $\boldsymbol{y} = \boldsymbol{f_1}[1 \ldots t']$ is a Lyndon word; we proceed to show that $\boldsymbol{y}$ is less than each of its proper suffixes. Let $\boldsymbol{w_k} = \boldsymbol{f_1}[t' - k + 1 \ldots t']$ for $1 \leq k < t'$. Since $\boldsymbol{f_1}$

is a Lyndon word and minimal in its conjugacy class, then $\boldsymbol{f_1}[1\,..\,k] \leq \boldsymbol{w_k}$; further, since $\beta(\boldsymbol{f_1})[t'] = 0$ we cannot have equality and so $\boldsymbol{f_1}[1\,..\,k] < \boldsymbol{w_k}$ which implies that $\boldsymbol{y} < \boldsymbol{w_k}$ as required.

In other words, we have shown that any border-free prefix of $\boldsymbol{f_1}$ is a Lyndon word and the result follows. □

**Corollary 6.3.2** *Any border-free prefix of a Lyndon word is a Lyndon word.*

**Lemma 6.3.3** Algorithm SNLB$f$P *is correct.*

*Proof.* (**Lemma 6.3.3**) In Step 2 we identify the factor $\boldsymbol{f_\mu}$ which starts with the letter $\mu < \gamma$. From Lyndon principles, Observation 6.2.4((2)),((3)), it follows that no factor to the left of $\boldsymbol{f_\mu}$ contains the letter $\mu$. Hence the prefix $\boldsymbol{s}[1\,..\,start(\boldsymbol{f_\mu})]$ is both border-free and non-Lyndon. However, it may not be the shortest one and so the algorithm continues to check through the prefix $\boldsymbol{p}$.

Now, consider the index $i$ of $\boldsymbol{p}$ computed in Step 4. Suppose that $i$ is an index of the factor $\boldsymbol{f_t}$; by Claim 6.3.1 we have $t > 1$. Let $k$ be the length of the prefix $\boldsymbol{p_t}$ of $\boldsymbol{f_t}$ that ends at $i$, and $\boldsymbol{p_1}$ be the prefix of $\boldsymbol{f_1}$ of length $k$. By the Lyndon factorization we have $\boldsymbol{p_1} \geq \boldsymbol{p_t}$ (in lexicographic order). If $\boldsymbol{p_1} = \boldsymbol{p_t}$ then this contradicts $\beta(\boldsymbol{p})[i] = 0$. Hence $\boldsymbol{p_1} > \boldsymbol{p_t}$ and so the prefix $\boldsymbol{s}[1\,..\,i]$ is both border-free and not a Lyndon word. Hence Algorithm SNLB$f$P correctly returns $\boldsymbol{s}[1\,..\,i]$. □

**Lemma 6.3.4** Algorithm SNLB$f$P *runs in* $\mathcal{O}(n)$ *time.*

*Proof.* [Proof of Lemma 6.3.4] Step 1 can be computed in $\mathcal{O}(n)$ time [Duv83, Lot05]. Step 2 applies an $\mathcal{O}(\log n)$ binary search. In Step 3 we compute the border array of the prefix $\boldsymbol{p}$ of $\boldsymbol{s}$. Clearly Steps 3 and 4 can be completed in $\mathcal{O}(n)$ time. Hence, the result follows. □

A binary Lyndon word can also be expressed in terms of Lyndon properties of the integer parameters (exponents) given by its *Run Length Encoding*. For a binary string $\boldsymbol{\ell}$, let $\mathcal{RLE}(\boldsymbol{\ell_a})$ denote the encoding (as a string) of the subsequence of $\boldsymbol{\ell}$ consisting of all letters $a$ but no letter $b$ $(p', p_1; \ldots; p_m)$, similarly $\mathcal{RLE}(\boldsymbol{\ell_b})$ denotes the encoding of the subsequence of $\boldsymbol{\ell}$ consisting of all letters $b$ $(q', q_1; \ldots; q_m)$.

**Lemma 6.3.5** *Let $\ell$ be a binary word with $\ell[1] = a$, $\ell[n] = b$ and associated encodings $\mathcal{RLE}(\ell_a)$ and $\mathcal{RLE}(\ell_b)$. Then $\ell$ is a Lyndon word if and only if, either*
*(i) $\mathcal{RLE}(\ell_a)$ is a Lyndon word on the alphabet $\{1 > 2 > 3 > \cdots\}$, or*
*(ii) $\mathcal{RLE}(\ell_a)$ is a repetition of a Lyndon word as in (i) and $\mathcal{RLE}(\ell_b)$ is a Lyndon word on the alphabet $\{1 < 2 < 3 < \cdots\}$.*

*Proof.* (**Lemma 6.3.5**) First we consider necessity. So suppose that (i) holds. Consider any rotation $\ell_a^r$ of $\ell_a$ (including those with split runs of $a$'s). Then $\mathcal{RLE}(\ell_a) < \mathcal{RLE}(\ell_a^r)$ in lexorder over $\{1 > 2 > 3 > \cdots\}$. Now suppose that (ii) holds. Then for a rotation $\ell_a^r$ of $\ell_a$, either $\mathcal{RLE}(\ell_a) < \mathcal{RLE}(\ell_a^r)$ in lexorder over $\{1 > 2 > 3 > \cdots\}$, or $\mathcal{RLE}(\ell_a) = \mathcal{RLE}(\ell_a^r)$ and $\mathcal{RLE}(\ell_b) < \mathcal{RLE}(\ell_b^r)$ in lexorder over $\{1 < 2 < 3 < \cdots\}$.

For sufficiency, the conditions guarantee that $\ell \in \mathcal{L}$. □

**Lemma 6.3.6** ***(Lyndon invalid point for border-free word)*** *Given a string $\ell \in \Sigma^n$, $n > 1$, such that $border(\ell) = 0$, if $\ell$ is not a Lyndon word ($\ell \notin \mathcal{L}$), then all the right extensions $\ell\ell'$ of $\ell$, such that $\ell' \in \Sigma^+$, are not Lyndon words either.*

*— We refer to this condition as the $\mathcal{L}$-fail condition and to the point (index) of where it occurs as the Lyndon invalid point.*

*Proof.* (**Lemma 6.3.6**) Since $\ell$ is border-free ($border(\ell) = 0$) but not a Lyndon word, then let $r$ be the rotation of $\ell$ which is the Lyndon word of the conjugacy class. Write $\ell = pq$ such that $r = qp$ and $qp < pq$.

**Case (1) -** If $|p| = |q|$, then since $r$ is border-free and a Lyndon word, $q \neq p$. Further, since $qp < pq$, we have $q < p$. It follows that $q\ell' p < pq\ell'$ and so $\ell\ell'$ cannot be a Lyndon word.

**Case (2) -** If $|q| > |p|$, then $r[1..|p|] < p$ (i.e., $q[1\,..\,|p|] < p$), we have $q < p$. It follows that $q\ell' p < pq\ell'$ and so $\ell\ell'$ cannot be a Lyndon word.

**Case (3) -** If $|q| < |p|$. We have $r = qp \in \mathcal{L}$, where $q = q_1 q_2 \cdots q_j$, $p = p_1 p_2 \cdots p_k$. We are required to show that $r' = pq\ell' \notin \mathcal{L}$; so suppose that $r' \in \mathcal{L}$. From $r$ we have that $q_1 \leq p_1$, while from $r'$ we have $p_1 \leq q_1$, which together implies $q_1 = p_1$. From the rotation of $r$ starting $p_1 p_2$ we have $q_2 \leq p_2$,

while from the rotation of $\boldsymbol{r'}$ starting $\boldsymbol{q_1 q_2}$ we find that $\boldsymbol{p_2} \leq \boldsymbol{q_2}$ giving $\boldsymbol{q_2} = \boldsymbol{p_2}$. We continue this argument for $|\boldsymbol{q}|$ elements which shows that the given word $\boldsymbol{pq}$ has a border.

In the first two cases the order is decided within the first $|\boldsymbol{p}|$ elements. □

**Lemma 6.3.7** ***(Lyndon invalid point for bordered word)*** *Given a string $\boldsymbol{\ell} \in \Sigma^n$, $n > 1$, such that $border(\boldsymbol{\ell}) > 0$, let $\{\boldsymbol{\ell'^1}, \boldsymbol{\ell'^2}, \boldsymbol{\ell'^3} \cdots\}$ be right extensions of $\boldsymbol{\ell}$ by $\{1, 2, 3, \cdots\}$ characters in $\Sigma$ respectively. Then $\boldsymbol{\ell}$ and all its right extensions are not Lyndon words if $\boldsymbol{\ell'^1}[|\boldsymbol{\ell'^1}|] < \boldsymbol{\ell'^1}[border(\boldsymbol{\ell}) + 1]$.*

*— Similarly to Lemma 6.3.6, we refer to this condition as the $\mathcal{L}$-fail condition and to the point (index) of where it occurs as the Lyndon invalid point.*

*Proof.* (**Lemma 6.3.7**) The lemma follows from the following two cases:

(1) $\boldsymbol{\ell} \notin \mathcal{L}$, this is immediate from the hypothesis that $border(\boldsymbol{\ell}) > 0$ and Observation 6.2.4(1).

(2) Consider the right extension $\boldsymbol{\ell'^m}$ of $\boldsymbol{\ell}$, where $m \geq 1$. Then the suffix $\boldsymbol{\ell'^m}[n - border(\boldsymbol{\ell}) + 1 \,..\, n + m]$ of $\boldsymbol{\ell'^m}$ is lexicographically less than $\boldsymbol{\ell'^m}[1 \,..\, border(\boldsymbol{\ell}) + 1]$ and consequently less than $\boldsymbol{\ell'^m}$, contradicting the property that a Lyndon word is strictly smaller than any of its proper suffixes. Hence no right extension of $\boldsymbol{\ell}$ is a Lyndon word.

In case (2) the order is decided within the first $border(\boldsymbol{\ell}) + 1$ elements. □

**Fact 6.3.8** *For a given string $\boldsymbol{s} \in \Sigma^n$, suppose we have computed $\beta(\boldsymbol{s})$. Then, for $1 \leq i \leq n$, the following holds true:*

$$\mathcal{L}\beta(\boldsymbol{s})[i] \in \{\beta(\boldsymbol{s})[i],\ \beta(\boldsymbol{s})[\beta(\boldsymbol{s})[i]],\ \beta(\boldsymbol{s})[\beta(\boldsymbol{s})[\beta(\boldsymbol{s})[i]]],\ \ldots,\ 0\}.$$

**Definition 32** *Consider a bounded alphabet $\Sigma$, for a given string $\boldsymbol{s} \in \Sigma^n$, we denote by $\Psi(\boldsymbol{s})$ the list (array) of length $n$ that is defined as: the $i$-th element $\Psi(\boldsymbol{s})[i] = true$ if the prefix $\boldsymbol{s}[1 \,..\, i]$ is a Lyndon word, otherwise $\Psi(\boldsymbol{s})[i] = false$, for $1 \leq i \leq n$.*

## 6.4 Lyndon Border Array Computation

In this section we develop an efficient algorithm for computing the Lyndon Border Array. We first recall an interesting relation that exists for borders which we refer to as the *Chain of Borders* henceforth. Since every border of any border of $\boldsymbol{s}$ is also a border of $\boldsymbol{s}$ (Observation 6.2.5 (2)), it turns out that, the border array $\beta(\boldsymbol{s})$ compactly describes all the borders of every prefix of $\boldsymbol{s}$. For every prefix $\boldsymbol{s}[1\,..\,i]$ of $\boldsymbol{s}$, the following sequence

$$\beta(\boldsymbol{s})^1[i], \beta(\boldsymbol{s})^2[i], \ldots, \beta(\boldsymbol{s})^m[i] \tag{6.1}$$

is well defined and monotonically decreasing to $\beta(\boldsymbol{s})^m[i] = 0$ for some $m \geq 1$ and this sequence identifies every border of $\boldsymbol{s}[1\,..\,i]$. Here, $\beta(\boldsymbol{s})^k[i]$ is the length of the $k$-th longest border of $\boldsymbol{s}[1\,..\,i]$, for $1 \leq k \leq m$. Sequence (6.1) identifies the above-mentioned chain of borders. We will also be using the usual notion of the length of the chain of borders. Clearly, the length of the chain in Sequence (6.1) is $m$. Also we use the following notion and notations. In Sequence (6.1), we call $\beta(\boldsymbol{s})^m[i] = 0$ the *last* value and $\beta(\boldsymbol{s})^{m-1}[i]$ the *penultimate* value in the chain. We now present the following interesting facts that will be useful in our algorithm.

**Fact 6.4.1** *The length of the chain of borders for a Lyndon Border is at most* 2.

*Proof.* The result follows, because, if a border is a Lyndon Word, then it cannot itself have a border (Observation 6.2.4 (1)).

□

**Fact 6.4.2** *Suppose* $\mathcal{L}\beta(\boldsymbol{s})[i] = x$. *Then* $\mathcal{L}\beta(\boldsymbol{s})[x] = 0$.

*Proof.* Immediate from Fact 6.4.1. □

Now we are ready to propose a straightforward naive algorithm to compute the Lyndon Border Array $\mathcal{L}\beta(\boldsymbol{s})$ for $\boldsymbol{s}[1\,..\,n]$ as follows.

The correctness of the algorithm follows directly from Facts 6.4.1 and 6.4.2. Now we discuss an efficient implementation of the algorithm. When we traverse through the chain of borders, we reach the penultimate value $k$ and then the last value $0$. Clearly,

1: Compute $\beta(\boldsymbol{s})[1\,..\,n]$
2: **for** $i = n \rightarrow 1$ **do**
3: Find the penultimate value $k$ of the chain of borders for $\beta(\boldsymbol{s})[i]$
4: **if** $\boldsymbol{s}[1\,..\,k]$ is a Lyndon word **then**
5: Set $\mathcal{L}\beta(\boldsymbol{s})[i] = k$
6: **else**
7: Set $\mathcal{L}\beta(\boldsymbol{s})[i] = 0$

Figure 6.1: Algorithm *Naive Lyndon Border Array Construction*

all we need is to check for each such chain, whether $\boldsymbol{s}[1\,..\,k]$ is a Lyndon word. So, we are always interested in finding whether $\boldsymbol{s}[1\,..\,k]$ is a Lyndon word where $\beta(\boldsymbol{s})[k] = 0$. At this point the computation of SNLB$f$P (Section 6.3) will be applied. To give an example, by Corollary 6.3.2, we know that any border-free prefix to the left of SNLB$f$P is a Lyndon word and any border-free prefix to the right of SNLB$f$P is non-Lyndon. This gives us an efficient weapon to check the *If* statement of Line 4 of the above algorithm. Finally, as can be seen below, we can make use of a stack data structure along with some auxiliary arrays to efficiently implement the above algorithm. In particular, we simply keep an array $Done[1\,..\,n]$ initially all false, using a stack and the SNLB$f$P index (say $r$). The algorithm is presented in Figure 6.2.

Clearly, the time complexity of the above algorithm depends on how many times a chain of borders is traversed. If we can ensure that a chain of borders is never traversed more than once, then the algorithm will surely be linear. To achieve that we use another array $\mathcal{P}val[1\,..\,n]$, initially all set to $-1$. This is required to efficiently compute the penultimate value of a chain of borders. The difficulty here arises because we may need to traverse a part of a chain of borders more than once through different indices because two different indices of the border array, $\beta(\boldsymbol{s})$, may have the same value. This may incur more cost and make the algorithm super-linear. To avoid traversing any part of a chain of borders more than once we use the array $\mathcal{P}val[1\,..\,n]$ as follows. Clearly, we only need to traverse the chain of borders to compute the penultimate value. So, as soon as we have computed the penultimate value $k$, for a chain, we store the value in the corresponding indices of $\mathcal{P}val$. To give an example, suppose we are considering the chain of borders $\beta(\boldsymbol{s})^1[i], \beta(\boldsymbol{s})^2[i], \ldots, \beta(\boldsymbol{s})^{m-1}[i], \beta(\boldsymbol{s})^m[i]$, where $\beta(\boldsymbol{s})^m[i] = 0$, and suppose that the penultimate value is $k$, i.e., $\beta(\boldsymbol{s})^{m-1}[i] = k$. Now suppose further that the indices involved in the above chain of borders are $i = i_1, i_2, \ldots, i_{m-1}, i_m$.

1: **for** $i = 1 \rightarrow n$ **do**
2: Set $Done[i]$ =FALSE
3: Compute $\beta(\boldsymbol{s})[1\,..\,n]$
4: Compute SNLB$f$P using Algorithm SNLB$f$P. Say, $\boldsymbol{s}[1\,..\,r]$ is the SNLB$f$P.
5: **for** $i = n \rightarrow 1$ **do**
6: **if** $Done[i]$ = TRUE **then**
7: continue ▷ i.e., skip what is done below
8: Compute the chain of $\beta(\boldsymbol{s})[i]$ and push each onto a stack $\mathcal{S}$.
9: Compute the penultimate value $k$ of the chain of $\beta(\boldsymbol{s})[i]$.
10: **while** $\mathcal{S}$ is nonempty **do**
11: Pop the value $j$ from the stack
12: **if** $k < r$ **then** ▷ i.e., $\boldsymbol{s}[1\,..\,k]$ is a Lyndon word
13: $\beta(\boldsymbol{s})[j] = k$
14: **else**
15: $\beta(\boldsymbol{s})[j] = 0$
16: $Done[j]$ =TRUE

Figure 6.2: Algorithm *Efficient Lyndon Border Array Construction*.

Then as soon as we have got the penultimate value, we update $\mathcal{P}val[i_1] = \mathcal{P}val[i_2] = .. = \mathcal{P}val[i_{m-1}] = k$. How does this help? If the same chain or part thereof is reached for computing the penultimate value, we can easily return the value from the corresponding $\mathcal{P}val[1\,..\,n]$ entry, and thus we never need to traverse a chain or part thereof more than once. This ensures the linear running time of the algorithm.

**Theorem 6.4.3** *For any given string $\boldsymbol{s}$ of length $n$, Algorithm* Efficient Lyndon Border Array Construction *computes* Lyndon Border Array *in $\mathcal{O}(n)$ time and linear space.*

## 6.5 Lyndon Suffix Array Computation

The well-known suffix array of a string records the lexicographically sorted list of all of its suffixes. Our next contribution is to show how the *Lyndon Suffix Array*, like the original suffix array, can be constructed in linear time; for a string of length $n$ it follows that the indexes in the Lyndon variant will be a subset of $\{1, 2, \ldots, n\}$. We will exploit the elegant fact that Lyndon suffixes are nested:

**Fact 6.5.1** *If the given string $\boldsymbol{s}$ is a Lyndon word, by Lyndon properties of Lyndon suffixes (Observation 6.2.4(4)), the indexes in the Lyndon Suffix Array will necessarily be increasing.*

In order to efficiently construct the Lyndon suffix array we could directly modify the linear-time and space efficient method of Ko and Aluru [KA03] given for the original data structure – this would involve lex-extension ordering (lexorder for substrings, see [DS14]) along with Fact 6.5.1. We note that the Ko-Aluru method has also recently been adapted to non-lexicographic $V$-order and $V$-letters, and applied in a novel Burrows-Wheeler transform [DS14], and hence is quite a versatile technique.

Let an *$L$-letter* $\boldsymbol{\ell} = \ell_1\ell_2..\ell_m$ substring denote the simple case of a Lyndon word such that $\ell_1 < \ell_i$ for $2 \leq i \leq m$, assumed to be of maximal length, that is, $\ell_1 = \ell_{m+1}$ (if $\ell_{m+1}$ exists); hence $|\boldsymbol{\ell}| \geq 1$.

Since, apart from the last letter, there may not be Lyndon suffixes of a string, we perform a linear scan to record the locations of the minimal letter $\ell_1$, say, in the string $\boldsymbol{s}$. Observe also that either an $L$-letter is a Lyndon suffix, or it is the prefix of a Lyndon suffix - the point is that an $L$-letter is a well-defined chunk of text, a substring of the input $\boldsymbol{s}$, as opposed to the classic single letter approach. In order to sort chunks of text lexicographically, we will apply *lex-extension* order defined as follows .

**Definition 33** *Suppose that according to some factorization $\mathcal{F}$, two strings $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^+$ are expressed in terms of nonempty factors: $\boldsymbol{u} = u_1u_2\cdots u_m, \boldsymbol{v} = v_1v_2\cdots v_n$. Then $\boldsymbol{u} <_{LEX(\mathcal{F})} \boldsymbol{v}$ if and only if one of the following holds:*

*(i) $\boldsymbol{u}$ is a proper prefix of $\boldsymbol{v}$ (that is, $u_i = v_i$ for $1 \leq i \leq m < n$); or*

*(ii) for some $i \in 1..min(m,n), u_j = v_j$ for $j = 1, 2, .., i-1$, and $u_i < v_i$ (in lexicographic order).*

First, using a linear scan we apply the $\mathcal{L}\beta$ to record the indexes of all the Lyndon suffixes of $\boldsymbol{s}$. The factorization $\mathcal{F}$ which we will use is that of decomposing the input into substrings of $L$-letters; in the case of unit length $L$-letters we concatenate them so that $\mathcal{F}$ has the general form:

$$\boldsymbol{u}\ell_1^{i_1}\boldsymbol{\ell_1}\ell_1^{i_2}\boldsymbol{\ell_2}\cdots\ell_1^{i_t}\boldsymbol{\ell_t}$$

where $\boldsymbol{u} \in \Sigma^*$ does not contain $\ell_1$, $t \geq 1$, each $i_j \geq 0$, every $\boldsymbol{\ell_j}$ is an $L$-letter and $|\boldsymbol{\ell_t}| \geq 1$ – in practice this just entails keeping track of occurrences of the letters $\boldsymbol{\ell}_1$ in

$\boldsymbol{s}$. Since we are computing Lyndon suffixes we can ignore $\boldsymbol{u}$ and hence assume that $\mathcal{F}$ has the form $\ell_1^{i_1}\boldsymbol{\ell_1}\ell_1^{i_2}\boldsymbol{\ell_2}\cdots\ell_1^{i_t}\boldsymbol{\ell_t}$.

An $\ell$-suffix is a suffix of $\mathcal{F}$ commencing with $\ell_1^{i_j}$. We now apply the Ko-Aluru linear method, consisting of three main steps, to the $\ell$-suffixes: which we first outline followed by further detail for each.

- Using a linear scan of the input string $\boldsymbol{s}$ and lex-extension ordering, divide all $\ell$-suffixes of $\boldsymbol{s}$ into two types: those Smaller ($S$) and those Larger ($L$) than their right-hand adjacent $\ell$-suffix. We assume that $|S| \leq |L|$, although the method also holds otherwise. That is, let $\boldsymbol{s_i}$ denote the suffix starting at index $i$, so $\boldsymbol{s_i} = \boldsymbol{s}[i \ldots n]$. Then the type $S$ Lyndon suffixes are the set $\{\boldsymbol{s_i} | \boldsymbol{s_i} < \boldsymbol{s_{i+1}}\}$ and the type $L$ Lyndon sufixes are the set $\{\boldsymbol{s_j} | \boldsymbol{s_j} > \boldsymbol{s_{j+1}}\}$.

- Sort by lex-extension all $\ell$-suffixes of type $S$ in $O(n)$-time using a modified Bucket Sort followed by recursion on at most half of the string.

- Using a linear scan obtain the lex-extension order of all remaining $\ell$-suffixes (assumed to be type $L$) from the sorted ones. This step is obtained from observing that the type $L$ $\ell$-suffixes occurring in $\boldsymbol{s}$ between two type $S$ $\ell$-suffixes, $S_i$ and $S_j$ where $i < j$, are already ordered such that $L_l <_{LEX(\mathcal{F})} L_k$ for $i < k < l < j$.

At this stage we have computed an $\ell$-suffix array. There are two final steps for computing the Lyndon suffix array. Firstly, the classic suffix array for the last $L$-letter $\boldsymbol{\ell_t}$ (without the prefix $\ell_1$) is processed directly using the Ko-Aluru method and the indexes inserted into the $\ell$-suffix array. Then we perform a linear scan of the $\ell$-suffix array, and applying Fact 6.5.1, by selecting only the sequence of increasing integers yields the Lyndon suffix array.

The last suffix is both type $S$ and $L$. The modification from lexicographic to lex-extension order follows from, firstly the linear factorization of the input into $L$-letters, and secondly that lex-extension ordering applies lexicographic order pairwise to $L$-letter substrings which each requires no more than time linear in the length of the $L$-letters – hence $\mathcal{O}(n)$ overall. The space efficiency follows from the original method [KA03].

### 6.5.1 A Simpler algorithm for computing a Lyndon Suffix Array from a Suffix Array

We present an alternative simple algorithm, derived from the classic suffix array, which also exploits the nested structure expressed in Fact 6.5.1. Suppose we are given the suffix array of the string $\boldsymbol{s}$. Now our algorithm finds the largest suffix (max), and then searches inside it to find the second largest one and so on, taking advantage of the fact that the suffixes are already sorted in the suffix array. Repeatedly finding the max value can be implemented efficiently using the Range Minimum Query (RMQ) [BFC00], which requires $\mathcal{O}(n)$ time pre-processing and then $\mathcal{O}(1)$ time for each query.

**Lemma 6.5.2 (Maximum range suffixes are Lyndon words)** *Suppose we are given the suffix array $\mathcal{A}$ of a string $\boldsymbol{s}$. Let the set $\mathcal{M}$ be the set of maximum values of the range of suffixes $\mathcal{A}[0\,..\,i]$, where $i$ is the index of the $i$-th suffix $\ell_i$. Each suffix $\ell \in \mathcal{M}$ is a Lyndon word.*

*Proof.* (**Lemma 6.5.2**) Suppose $\ell$ is a max range suffix ($\ell \in \mathcal{M}$) at index $i$ with order value $SA[i] = m$. Suppose there exists a suffix $\boldsymbol{\ell''}$ of $\boldsymbol{\ell}$ at index $i''$ (w.r.t. to $\mathcal{A}$) with order value $\mathcal{A}[i''] = m''$, and also suppose $\boldsymbol{\ell''} < \boldsymbol{\ell}$. Hence $m'' > m$ ($\mathcal{A}[i''] > \mathcal{A}[i]$) where $i'' < i$, which contradicts the fact that $m$ is the maximum value in the range $\mathcal{A}[0\,..\,i]$. Therefore, $\boldsymbol{\ell}$ is strictly smaller than all of its proper suffixes – we conclude that $\ell$ is a Lyndon word. □

The linear-time method for computing the Lyndon suffix array is very simple. Below, we first outline the steps followed by the pseudo-code.

1. Compute the suffix array $\mathcal{A}$ of $\boldsymbol{s}\$$.
2. Find the value $max = \text{MAX}(\mathcal{A}[0\,..\,n])$ and its index $i$ in the suffix array, then add the value $max$ to Lyndon Suffix array $\mathcal{L}S$.
3. Find the value $max$ and its index $i$ in the range $\mathcal{A}[0\,..\,i]$, $max = \text{MAX}(\mathcal{A}[0\,..\,i])$ and $i = \text{Indexof}(max, \mathcal{A})$, then add the value $max$ to $\mathcal{L}S$.
4. Repeat step 2 until the set $\mathcal{A}[0\,..\,i]$ is empty.

**procedure** COMPUTELSA($\boldsymbol{s}$)
$n \leftarrow |s|;\ \mathcal{L}[1\,..\,n] \leftarrow (-1)^n$
▷ *compute the suffix array of* $\boldsymbol{s}\$$
$\mathcal{A} \leftarrow SA(\boldsymbol{s}\$)$
$i \leftarrow n; j \leftarrow n; max \leftarrow 0$
**while** $(\mathcal{A}[0\,..\,i] \neq \emptyset)$ **do**
▷ *find the maximum value (and its index) in the range* $\mathcal{A}[0\,..\,i]$.
$(max, i) \leftarrow (\mathcal{A}[idx], idx) \mid idx = \arg\max_{idx}(A[idx])$ for $0 \leq idx < i$
$j \leftarrow j - 1$
$\mathcal{L}[j] \leftarrow max$
return $\mathcal{L}$

Figure 6.3: Computing the Lyndon Suffix Array from a Suffix Array.

**Theorem 6.5.3** *For any given string* $\boldsymbol{s}$ *of length* $n$*, Algorithm* $\boldsymbol{ComputeLSA}$ *computes the* Lyndon Suffix Array *of string* $\boldsymbol{s}$ *from its Suffix Array in* $\mathcal{O}(n)$ *time and space.*

# Article: # 7

# Simple Linear Comparison of Strings in $V$-order

In this article, we focus on a total (but non-lexicographic) ordering of strings called $V$-order. We devise a new linear-time algorithm for computing the $V$-comparison of two finite strings. In comparison with the previous algorithm in the literature, our algorithm is both conceptually simpler, based on recording letter positions in increasing order, and more straightforward to implement, requiring only linked lists.

---

## 7.1 Introduction

An important task required in many combinatorial computations is deciding the relative order of two members of a totally ordered set [KS99, Rus03], for instance organizing words in a natural language dictionary. Binary comparison of finite strings (words) thus arises as a primitive operation, a building block, in more complex procedures, which therefore requires efficient implementation.

We first discuss some known techniques for totally ordering sets, and then introduce our contribution: a new linear string comparison algorithm using $V$-order.

Given an integer $n \geq 1$ and a nonempty set of symbols $\Sigma$ (bounded or unbounded), a *string* $\boldsymbol{x}$ (written in math bold) of *length* $n$ is a sequence of *letters* $x_1, x_2, \ldots, x_n$ (written in regular math mode), with each $x_i \in \Sigma$. We will often represent $\boldsymbol{x}$ as an array $\boldsymbol{x}[1..n]$ with entries $\boldsymbol{x}[1], \boldsymbol{x}[2], \ldots, \boldsymbol{x}[n]$. The classic and commonly used method for organizing sets of strings is lexicographic (dictionary) order. Formally, if $\Sigma$ is a totally ordered alphabet then *lexicographic ordering* (lexorder) $\boldsymbol{u} < \boldsymbol{v}$ with $\boldsymbol{u}, \boldsymbol{v} \in \Sigma^+$ is defined if and only if either $\boldsymbol{u}$ is a proper prefix of $\boldsymbol{v}$, or $\boldsymbol{u} = \boldsymbol{r}a\boldsymbol{s}$, $\boldsymbol{v} = \boldsymbol{r}b\boldsymbol{t}$ for some $a, b \in \Sigma$ such that $a < b$ and for some $\boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t} \in \Sigma^*$.

Lexorder is a very natural method for deciding precedence and organizing information which also finds many uses in computer science, typically in constructing data structures and related applications:

- Building indexes for information retrieval, particularly self-indexes which replace the text and support almost optimal space and search time [NM07].
- Constructing suffix arrays, which record string suffix starting positions in the lexorder of the suffixes, and thus support binary search [KA03, KSB06, NZC09].
- The Burrows-Wheeler Transform (BWT), which applies suffix sorting, and exhibits data clustering properties, hence is suitable for preprocessing data prior to compression activities [ABM08, CDP05].
- The application of automata for bioinformatics sequence alignment. The BWT is extended for finite automata representing the multiple alignment problem - the paths in the automaton are sorted into lexorder thus extending the suffix sorting framework related to the classic BWT [SVM11].

- An important class in the study of combinatorics on words is *Lyndon words* [Lot83] – strings (words) which are lexicographically least amongst the cyclic rotations of their letters (characters) – see also [Smy03]; furthermore, any string can be uniquely factored into Lyndon words [CFL58] – Duval's algorithm cleverly detects the lexicographic order between factors in linear time [Duv83, Day11]. The Lyndon decomposition allows for efficient 'divide-and-conquer' of a string into patterned factors; numerous applications include periodic musical structures [Che04], string matching [BGM11, CP91], and algorithms for digital geometry [BLPR09].

- Hybrid Lyndon structures, introduced in [DDS13], based on two methods of ordering strings one of which is lexorder.

Naively, lexorder $\boldsymbol{u} < \boldsymbol{v}$ can be decided in time linear in the length of the shorter string, and space linear in the length of the longer string; various data structures may be used for enhancing this string comparison. In [DIS94] the Four Russians technique [IS92] is proposed to compare strings of length $n$ on a bounded alphabet in $\mathcal{O}(1)$ time, while for an unbounded alphabet the parallel construction of a merged suffix tree using the CRCW PRAM model [IS92] is proposed that can be constructed in $\mathcal{O}(\log n \log\log n)$ time using $\mathcal{O}(n/\log n)$ processors; using this tree, sequential comparison requires $\mathcal{O}(\log\log n)$ time.

A class of lexorder-type total orders is easily obtained from permuting the usual order $1, 2, \ldots n$ of pairwise comparison of letters, along with interchanging $<$ with $>$ and so on; for example *relex order* (reverse lexicographic) [Rus03], and *co-lexorder* (lexorder of reversed strings) studied and applied to string factorization in [DEDS09].

Non-lexicographic methods include deciding precedence by minimal change such as Gray's *reflected binary code*, where two successive values differ in only one bit, hence well-suited for error correction in digital communications [Gra53, Sav96]. A more recent example is $V$*-order* [Day85, DD96, DD97] which is the focus of this paper: we first introduce this technical method for comparing strings and then consider it algorithmically.

Let $\Sigma$ be a totally ordered alphabet, and let $\boldsymbol{u} = u_1u_2...u_n$ be a string over $\Sigma$. Define $h \in \{1, \ldots, n\}$ by $h = 1$ if $u_1 \leq u_2 \ldots \leq u_n$; otherwise, by the unique value such that $u_{h-1} > u_h \leq u_{h+1} \leq u_{h+2} \leq \ \ldots \leq u_n$. Let $\boldsymbol{u}^* = u_1u_2...u_{h-1}u_{h+1}...u_n$,

where the star * indicates deletion of the letter $u_h$. Write $\boldsymbol{u}^{s*}$ for $(...(\boldsymbol{u}^*)^*...)^*$ with $s \geq 0$ stars [1]. Let $g = \max\{u_1, u_2, ..., u_n\}$, and let $k$ be the number of occurrences of $g$ in $\boldsymbol{u}$. Then the sequence $\boldsymbol{u}, \boldsymbol{u}^*, \boldsymbol{u}^{2*}, ...$ ends $g^k, ..., g^2, g^1, g^0 = \varepsilon$. In the *star tree* each string $\boldsymbol{u}$ over $\Sigma$ labels a vertex, and there is a directed edge from $\boldsymbol{u}$ to $\boldsymbol{u}^*$, with the empty string $\varepsilon$ as the root.

**Definition 34** *We define $V$-order $\prec$ between distinct strings $\boldsymbol{u}, \boldsymbol{v}$. First $\boldsymbol{v} \prec \boldsymbol{u}$ if $\boldsymbol{v}$ is in the path $\boldsymbol{u}, \boldsymbol{u}^*, \boldsymbol{u}^{2*}, ..., \varepsilon$. If $\boldsymbol{u}, \boldsymbol{v}$ are not in a path, there exist smallest $s, t$ such that $\boldsymbol{u}^{(s+1)*} = \boldsymbol{v}^{(t+1)*}$. Put $\boldsymbol{c} = \boldsymbol{u}^{s*}$ and $\boldsymbol{d} = \boldsymbol{v}^{t*}$; then $\boldsymbol{c} \neq \boldsymbol{d}$ but $|\boldsymbol{c}| = |\boldsymbol{d}| = m$ say. Let $j$ be the greatest $i$ in $1 \leq i \leq m$ such that $\boldsymbol{c}[i] \neq \boldsymbol{d}[i]$. If $\boldsymbol{c}[j] < \boldsymbol{d}[j]$ in $\Sigma$ then $\boldsymbol{u} \prec \boldsymbol{v}$. Clearly $\prec$ is a total order.*

**Example 7.1.1** *Over the binary alphabet with $0 < 1$: in lexorder, 0101 $<$ 01110; in $V$-order, 0101 $\prec$ 01110.*
*Over the naturally ordered integers: in lexorder, 123456 $<$ 2345; in $V$-order, 2345 $\prec$ 123456.*
*Over the naturally ordered Roman alphabet: in lexorder, $eabecd < ebaedc$; in $V$-order, $ebaedc \prec eabecd$.*

String comparison in $V$-order $\prec$ was first considered algorithmically in [DDS11, DDS13] – the dynamic longest matching suffix of the pair of input strings, together with a doubly-linked list which simulated letter deletions and hence paths in the star tree, enabled deciding order; these techniques achieved $V$-comparison in worst-case time and space proportional to string length – thus asymptotically the same as naive comparison in lexorder.

Currently known applications of $V$-order, utilizing linear-time $V$-comparison, and generally derived from lexorder or Lyndon cases are as follows:

- A $V$-order structure, an instance of a hybrid Lyndon word and known as a $V$-word [DD03], similarly to the classic Lyndon case, gives an instance of an African musical rhythmic pattern [CT03].

[1] Note that this star operator, as defined in [DD96], [DD03] etc, is distinct from the Kleene star operator: Kleene star is applied to sets, while this $V$-star is applied to strings.

- Linear factorization of a string into factors ($V$-words) sequentially [DDS11] and in parallel [DDIS13] – yielding factors which are distinct from the Lyndon factorization of the given string [DDS13].
- Modification of a linear suffix array construction [KA03] from lexorder to $V$-order [DS14] thus allowing efficient $V$-ordering of the cyclic rotations of a string.
- Applying the above suffix array modification to compute a novel Burrows-Wheeler transform ($V$-BWT) using, not the usual lexorder, but rather $V$-order, in $\theta(n)$ time and space, when it is known that the input string is a $V$-word [DS14] – achieving instances of enhanced data clustering.

These initial avenues suggest that further uses of $V$-order, analogous to the practical functions listed for lexorder and Lyndon words, will continue to arise, including for instance those for suffix trees – thus necessitating efficient implementations of the primitive $V$-comparison.

We introduce here a new algorithm for computing the $V$-comparison of two finite strings – the advantage is that it is both conceptually simpler, based on recording letter positions in increasing order, and more straightforward to implement, requiring only linked lists. The time complexity is $\mathcal{O}(n + |\Sigma|)$ and similarly the space complexity is $\mathcal{O}(n + |\Sigma|)$. However, in computational practice the alphabet, like the input, can be assumed to be finite - at most $\mathcal{O}(n)$ – and so the algorithm runs in essentially linear time.

## 7.2 $V$-order String Comparison Algorithm

In this section, we present a novel linear-time algorithm for $V$-order string comparison. Before going into the algorithmic details, we present relevant definitions and results from the literature useful in describing and analyzing our algorithm, starting with a unique representation of a string.

**Definition 35** *([DD03, DDS11, DDS13]) The **V-form** of a string $\boldsymbol{x}$ is defined as*

$$V_k(\boldsymbol{x}) = \boldsymbol{x} = \boldsymbol{x_0} g \boldsymbol{x_1} g \cdots \boldsymbol{x_{k-1}} g \boldsymbol{x_k}$$

*for strings $\boldsymbol{x_i}$, $i = 0, 1, \ldots, k$, where $g$ is the largest letter in $\boldsymbol{x}$ – thus we suppose that $g$ occurs exactly $k$ times. For clarity, when more than one string is involved, we will use the notation $g = \mathcal{L}_{\boldsymbol{x}}$.*

The following lemma is the key to our algorithm.

**Lemma 7.2.1** *([DD96, DD03, DDS11, DDS13]) Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with the corresponding V-forms as follows:*

$$\boldsymbol{v} = \boldsymbol{v}_0\mathcal{L}_{\boldsymbol{v}}\boldsymbol{v}_1\mathcal{L}_{\boldsymbol{v}}\boldsymbol{v}_2 \cdots \boldsymbol{v}_{j-1}\mathcal{L}_{\boldsymbol{v}}\boldsymbol{v}_j$$

$$\boldsymbol{x} = \boldsymbol{x}_0\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_1\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_2 \cdots \boldsymbol{x}_{k-1}\mathcal{L}_{\boldsymbol{x}}\boldsymbol{x}_k$$

*Let $h \in \{0 \ldots \max(j, k)\}$ be the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. Then $\boldsymbol{v} \prec \boldsymbol{x}$ if, and only if, one of the following conditions holds:*

1. $\mathcal{L}_{\boldsymbol{v}} < \mathcal{L}_{\boldsymbol{x}}$
2. $\mathcal{L}_{\boldsymbol{v}} = \mathcal{L}_{\boldsymbol{x}}$ *and* $j < k$
3. $\mathcal{L}_{\boldsymbol{v}} = \mathcal{L}_{\boldsymbol{x}}$, $j = k$ *and* $\boldsymbol{v}_h \prec \boldsymbol{x}_h$.

**Lemma 7.2.2** *([DDS11, DDS13]) Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$. If $\boldsymbol{v}$ ($\boldsymbol{x}$ resp.) is a subsequence of $\boldsymbol{x}$ ($\boldsymbol{v}$ resp.) then $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$ resp.).*

We will use some simple data structures, which are initialized by preprocessing steps. We use $\texttt{Map}_u(a)$ to store, in increasing order, the positions of the character $a$ in a string $\boldsymbol{u}$. $\texttt{Map}_u(\Sigma)$ records the 'maps' of all $a \in \Sigma$. To construct $\texttt{Map}_u(\Sigma)$ we take an array of size $\Sigma$. For each $a \in \Sigma$, we construct a linked list that stores the positions $i \in [1..|\boldsymbol{u}|]$ in increasing order such that $\boldsymbol{u}[i] = a$.

**Example 7.2.3** *Suppose we have a string $\boldsymbol{u}$ as follows:*

| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{u}$ | = | 8 | 5 | 8 | 2 | 1 | 8 | 7 | 6 | 5 | 4 | 3 |

$\texttt{Map}_u(\Sigma)$ *is shown below for the string $\boldsymbol{u}$ defined above.*

$$
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
5 & 4 & 11 & 10 & 2 & 8 & 7 & 1 \\
 & & & & \downarrow & & & \downarrow \\
 & & & & 9 & & & 3 \\
 & & & & & & & \downarrow \\
 & & & & & & & 6
\end{array}
$$

This leads to the following lemma.

**Lemma 7.2.4** *Given a string $\boldsymbol{u}$ of length $n$ we can build $\texttt{Map}_u(\Sigma)$ in $\mathcal{O}(n + |\Sigma|)$ time and space.*

*Proof.* As mentioned above, we realize $\texttt{Map}_u(\Sigma)$ as an array of linked lists; in particular, we maintain a linked list $\texttt{Map}_u(a)$ for each $a \in \Sigma$. Initially, for all $a \in \Sigma$, $\texttt{Map}_u(a) = NULL$, i.e., each list is empty. We simply traverse the string $\boldsymbol{u}$, and at each position $i$ do the following. Assume that $\boldsymbol{u}[i] = \alpha$, then we append $i$ to the list $\texttt{Map}_u(\alpha)$. Clearly this simple algorithm is correct and runs in linear time, i.e., $O(n)$. Since, we need to initialize the $|\Sigma|$ size array, the total running time becomes $\mathcal{O}(n + |\Sigma|)$. The space requirement is also clearly $O(n + |\Sigma|)$. □

We will now prove a number of new lemmas that will be used in the string comparison algorithm – first we will introduce some notations. Let $firstMiss(\boldsymbol{u}, \boldsymbol{v})$ denote the first mismatch entry between $\boldsymbol{u}, \boldsymbol{v}$. More formally, we say $\ell = firstMiss(\boldsymbol{u}, \boldsymbol{v})$ if and only if $\boldsymbol{u}[\ell] \neq \boldsymbol{v}[\ell]$ and $\boldsymbol{u}[i] = \boldsymbol{v}[i]$, for all $1 \leq i < \ell$. In what follows, the notion of a global mismatch and a local mismatch is useful in the context of two strings $\boldsymbol{u}, \boldsymbol{v}$ and their respective substrings $\boldsymbol{u'}, \boldsymbol{v'}$. In particular, $firstMiss(\boldsymbol{u}, \boldsymbol{v})$ would be termed as the global mismatch in this context and $firstMiss(\boldsymbol{u'}, \boldsymbol{v'})$ would be termed as a local mismatch, i.e., local to the corresponding substrings. For this global/local notion, the context $\mathcal{C}$ is important and is defined with respect to the two strings and their corresponding substrings, i.e., the context here would be denoted by $\mathcal{C}\langle(\boldsymbol{u}, \boldsymbol{u'}), (\boldsymbol{v}, \boldsymbol{v'})\rangle$. Also, for the $V$-form of a string $\boldsymbol{u}$ we will use the following convention: $\mathcal{L}_{u,\ell}$ denotes the $\ell$-th $\mathcal{L}_u$ in the $V$-form of $\boldsymbol{u}$ and $pos(\mathcal{L}_{u,\ell})$ will be used to denote its index/position in $\boldsymbol{u}$. With this extended notation, the $V$-form of $\boldsymbol{u}$ can be rewritten as follows:

$$\boldsymbol{u} = \boldsymbol{u}_0 \, \mathcal{L}_{u,1} \, \boldsymbol{u}_1 \, \mathcal{L}_{u,2} \, \boldsymbol{u}_2 \, \cdots \, \boldsymbol{u}_{j-1} \, \mathcal{L}_{u,j} \, \boldsymbol{u}_j.$$

Moreover, within the context $\mathcal{C}$, the strings $\boldsymbol{u}, \boldsymbol{v}$ are referred to as the *superstrings* and $\boldsymbol{u'}, \boldsymbol{v'}$ as the *substrings*.

**Lemma 7.2.5** *Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with the corresponding $V$-forms as follows:*

$$\boldsymbol{v} = \boldsymbol{v}_0 \mathcal{L}_v \boldsymbol{v}_1 \mathcal{L}_v \boldsymbol{v}_2 \cdots \boldsymbol{v}_{j-1} \mathcal{L}_v \boldsymbol{v}_j$$

$$\boldsymbol{x} = \boldsymbol{x}_0 \mathcal{L}_x \boldsymbol{x}_1 \mathcal{L}_x \boldsymbol{x}_2 \cdots \boldsymbol{x}_{k-1} \mathcal{L}_x \boldsymbol{x}_k$$

*Assume that $\mathcal{L}_v = \mathcal{L}_x$ and $j = k$. Let $h \in \{0 \ldots \max(j,k)\}$ be the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. Now assume that $\ell_h = firstMiss(\boldsymbol{v}_h, \boldsymbol{x}_h)$ and $\ell_f = firstMiss(\boldsymbol{v}, \boldsymbol{x})$. In other words, $\ell_h$ is the index of the first mismatch entry between the substrings $\boldsymbol{v}_h, \boldsymbol{x}_h$, whereas $\ell_f$ is the index of the first mismatch entry between the two strings $\boldsymbol{v}$ and $\boldsymbol{x}$. Then we must have $\ell_f = \sum_{i=0}^{h-1}(|\boldsymbol{v}_i| + 1) + \ell_h$. (Or equivalently, $\ell_f = \sum_{i=0}^{h-1}(|\boldsymbol{x}_i| + 1) + \ell_h$).*

*Proof.* This lemma basically claims that the first mismatch of $\boldsymbol{v}_h$ and $\boldsymbol{x}_h$ would in fact be the first mismatch globally. We prove this by considering two cases.

**Case 1:** $\texttt{Map}_v(\mathcal{L}_v) = \texttt{Map}_x(\mathcal{L}_x)$**.** In this case, all the positions of $\mathcal{L}_v$ and $\mathcal{L}_x$ in $\boldsymbol{v}$ and $\boldsymbol{x}$ respectively are identical. So, definitely, the first global mismatch and the first local mismatch in the context $\mathcal{C}\langle(\boldsymbol{v}, \boldsymbol{v}_h), (\boldsymbol{x}, \boldsymbol{x}_h)\rangle$ would have to be identical. And hence the result follows.

$$\begin{array}{cccccccccccc}
- & - & v_{\ell-3} & \mathcal{L}_{v,\ell-2} & v_{\ell-2} & \mathcal{L}_{v,\ell-1} & < \cdots\cdots v_{\ell-1} \cdots\cdots > & \mathcal{L}_{v,\ell} & & & - & - \\
 & & & & & & \uparrow & & & & & \\
 & & & & & & \ell_f & & & & & \\
 & & & & & & \downarrow & & & & & \\
- & - & x_{\ell-3} & \mathcal{L}_{x,\ell-2} & x_{\ell-2} & \mathcal{L}_{x,\ell-1} & < \cdots\cdots\cdots x_{\ell-1} \cdots\cdots\cdots > & \mathcal{L}_{x,\ell} & & & - & -
\end{array}$$

Figure 7.1: The case when $\texttt{Map}_v(\mathcal{L}_v) \neq Map_x(\mathcal{L}_x)$

**Case 2:** $\texttt{Map}_v(\mathcal{L}_v) \neq \texttt{Map}_x(\mathcal{L}_x)$**.** It would be useful to follow the arguments with reference to Figure 7.1. Recall that according to our hypothesis, $\mathcal{L}_v = \mathcal{L}_x$, $j = k$. Since, $\texttt{Map}_v(\mathcal{L}_v)$ and $\texttt{Map}_x(\mathcal{L}_x)$ differ with each other, assume that the first difference occurs between $\mathcal{L}_{v,\ell}$ and $\mathcal{L}_{x,\ell}$, i.e., between the $\ell$-th entries. So, we have $\boldsymbol{v}[pos(\mathcal{L}_{v,\ell})] = \mathcal{L}_v = \mathcal{L}_x = \boldsymbol{x}[pos(\mathcal{L}_{x,\ell})]$. However, $pos(\mathcal{L}_{v,\ell}) \neq pos(\mathcal{L}_{x,\ell})$ and this is the first/least such position. Assume w.t.l.o.g. that $pos(\mathcal{L}_{v,\ell}) < pos(\mathcal{L}_{x,\ell})$.

Now, if the first global mismatch position, i.e., $\ell_f < pos(\mathcal{L}_{v,\ell-1})$, then the situation is identical to Case 1. So, let us assume that $\ell_f > pos(\mathcal{L}_{v,\ell-1})$. Note that we cannot have $\ell_f = pos(\mathcal{L}_{v,\ell-1})$, for otherwise we would have $\mathcal{L}_{v,\ell-1} \neq \mathcal{L}_{x,\ell-1}$, contradicting our assumption above. In this case, we must have $h = \ell - 1$, and clearly $\boldsymbol{v}[pos(\mathcal{L}_{v,\ell})] \neq \boldsymbol{x}[pos(\mathcal{L}_{v,\ell})]$. So, we must have $\ell_f \leq pos(\mathcal{L}_{v,\ell})$; further, the first global mismatch and the first local mismatch in the context $\mathcal{C}\langle(\boldsymbol{v}, \boldsymbol{v}_h), (\boldsymbol{x}, \boldsymbol{x}_h)\rangle$ would have to be identical, and hence the result follows. Note that we also have $|\boldsymbol{v}_h| < |\boldsymbol{x}_h|$. □

**Corollary 7.2.6** *If in Case 2 of Lemma 7.2.5 we have $\ell_f = pos(\mathcal{L}_{v,\ell})$, then $\boldsymbol{v}_h$ is a proper prefix of $\boldsymbol{x}_h$.*

Interestingly, we can extend Lemma 7.2.5 further if we consider the (inner) contexts within (outer) contexts as the following lemma shows. In other words $V$-form can be applied recursively and independently as shown in [DDS11]. In what follows, for given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with corresponding $V$-forms, the condition that $\mathcal{L}_v = \mathcal{L}_x$, $j = k$ will be referred to as $Cond\text{-}I(\boldsymbol{v}, \boldsymbol{x})$.

**Lemma 7.2.7** *Suppose we are given distinct strings $\boldsymbol{v}$ and $\boldsymbol{x}$ with corresponding $V$-forms, and assume that $Cond\text{-}I(\boldsymbol{v}, \boldsymbol{x})$ holds. Now consider the (outer) context $\mathcal{C}^0\langle(\boldsymbol{v}, \boldsymbol{v}_{h^0}), (\boldsymbol{x}, \boldsymbol{x}_{h^0})\rangle$, where $h^0$ is the least integer such that $\boldsymbol{v}_{h^0} \neq \boldsymbol{x}_{h^0}$.*

*Now similarly consider the $V$-forms of $\boldsymbol{v}_{h^0}$ and $\boldsymbol{x}_{h^0}$ and assume that $Cond\text{-}I(\boldsymbol{v}_{h^0}, \boldsymbol{x}_{h^0})$ holds. Further, consider the (inner) context $\mathcal{C}^1\langle(\boldsymbol{v}_{h^0}, \boldsymbol{v}_{h^1}), (\boldsymbol{x}_{h^0}, \boldsymbol{x}_{h^1})\rangle$, where $h^1$ is the least integer such that $\boldsymbol{v}_{h^1} \neq \boldsymbol{x}_{h^1}$.*

*Then the global mismatch of the context $\mathcal{C}^0$ coincides with the local mismatch of the context $\mathcal{C}^1$.*

*Proof.* Basically, the result follows by applying the arguments of Lemma 7.2.5 to form a chain of arguments between the inner and the outer contexts. □

**Corollary 7.2.8** *Given nested contexts $\mathcal{C}^i, 0 \leq i \leq k$ satisfying the hypotheses of Lemma 7.2.7, the global mismatch of context $\mathcal{C}^0$ coincides with the local mismatch of context $\mathcal{C}^k$.*

Corollary 7.2.8 establishes that the first global mismatch will always be the first mismatch as we go further within inner contexts through the chain of outer and inner contexts.

Now we can focus on the string comparison algorithm: Algorithm `CompareV`. (See Example 7.2.12 at the end of this section.) Suppose we are given two distinct strings $\boldsymbol{p}$ and $\boldsymbol{q}$, then the algorithm performs the following steps.

**Step 1: Preprocessing Step.** Compute $\texttt{Map}_p(\Sigma)$ and $\texttt{Map}_q(\Sigma)$. We also compute the first mismatch position $\ell_f$ between $\boldsymbol{p}$ and $\boldsymbol{q}$. This will be referred to as the global mismatch position and will be independent of any context within the iterations of the algorithm.

Then we repeat the following sub-steps in Step 2. During different iterations of the execution of these stages we will be considering different contexts by proceeding from outer to inner contexts. Initially, we will start with the outermost context, i.e., $\mathcal{C}^0\langle(\boldsymbol{p}, \boldsymbol{p}_{h^0}), (\boldsymbol{q}, \boldsymbol{q}_{h^0})\rangle$, where $h^0$ is the least integer such that $\boldsymbol{p}_{h^0} \neq \boldsymbol{q}_{h^0}$. At each iteration, we will be considering the largest $\alpha \in \Sigma$ that is present within one of the superstrings in the context. In other words, if the current context is $\mathcal{C}^0$, as is the case during the initial iteration, we will consider the largest $\alpha$ such that $\alpha \in \boldsymbol{p}$ or $\alpha \in \boldsymbol{q}$.

**Step 2:** Throughout this step we will assume that the current context is $\mathcal{C}\langle(\boldsymbol{v}, \boldsymbol{v}_h), (\boldsymbol{x}, \boldsymbol{x}_h)\rangle$, where $h$ is the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. So, initially we have $\mathcal{C} = \mathcal{C}^0$. Suppose we are now considering $\alpha \in \Sigma$, then it must be the largest $\alpha \in \Sigma$ such that either $\alpha \in \boldsymbol{v}$ or $\alpha \in \boldsymbol{x}$. We proceed to the following sub-steps:

**Step 2.a:** We compute $\texttt{Map}_v(\alpha)$ from $\texttt{Map}_p(\alpha)$ where $\texttt{Map}_v(\alpha)$ contains the positions that are only within the range of $\boldsymbol{v}$ in the current context $\mathcal{C}$. Similarly, we compute $\texttt{Map}_x(\alpha)$ from $\texttt{Map}_q(\alpha)$ where $\texttt{Map}_x(\alpha)$ contains the positions that are only

within the range of $\boldsymbol{x}$ in the current context $\mathcal{C}$. Now we compare $\texttt{Map}_v(\alpha)$ and $\texttt{Map}_x(\alpha)$, which yields two cases.

**Step 2.a.(i):** In this case, $\texttt{Map}_v(\alpha) = \texttt{Map}_x(\alpha)$.

This means that within the current context $\mathcal{C}$, considering the $V$-form of the superstrings $\boldsymbol{v}$ and $\boldsymbol{x}$, we must have $\mathcal{L}_v = \mathcal{L}_x$ and $j = k$. So, we need to check Condition 3 of Lemma 7.2.1. We identify $h$ such that $h$ is the least integer with $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. By Lemmas 7.2.5, 7.2.7 and Corollary 7.2.8 we know that this $h$ can be easily identified because it is identical to the global mismatch position $\ell_f$.

Then we iterate to Step 2 again with the inner context $\mathcal{C}^1\langle(\boldsymbol{v}_h, \boldsymbol{v}_{h^1}), (\boldsymbol{x}_h, \boldsymbol{x}_{h^1})\rangle$, where $h^1$ is the least integer such that $\boldsymbol{v}_{h^1} \neq \boldsymbol{x}_{h^1}$. In other words, we assign $\mathcal{C} = \mathcal{C}^1$ and then repeat Step 2 for $\beta \in \Sigma$ where $\beta < \alpha$.

**Step 2.a.(ii):** In this case, $\texttt{Map}_v(\alpha) \neq \texttt{Map}_x(\alpha)$.

[C1] If $\texttt{Map}_v(\alpha) = \emptyset$ ($\texttt{Map}_x(\alpha) = \emptyset$ resp.), we have Condition 1 of Lemma 7.2.1 satisfied ($\varepsilon$ is the least string in $V$-order) and hence we return $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$ resp.). Note that this effectively decides $\boldsymbol{p} \prec \boldsymbol{q}$ ($\boldsymbol{q} \prec \boldsymbol{p}$ resp.) and the algorithm terminates.

[C2] If $|\texttt{Map}_v(\alpha)| < |\texttt{Map}_x(\alpha)|$ ($|\texttt{Map}_x(\alpha)| < |\texttt{Map}_v(\alpha)|$ resp.), we have Condition 2 of Lemma 7.2.1 satisfied and hence we return $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$ resp.). Similarly, this effectively decides $\boldsymbol{p} \prec \boldsymbol{q}$ ($\boldsymbol{q} \prec \boldsymbol{p}$ resp.) and the algorithm terminates.

[C3] Otherwise, we have $\mathcal{L}_v = \mathcal{L}_x$ and $j = k$. So, we need to check Condition 3 of Lemma 7.2.1, and identify $h$ such that $h$ is the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. By Lemmas 7.2.5, 7.2.7 and Corollary 7.2.8 we know that $h$ can be easily identified because it is identical to the global mismatch position $\ell_f$. Now we do a final check as to whether $\boldsymbol{v}_h$ is a subsequence (in fact, a prefix) of $\boldsymbol{x}_h$ according to Corollary 7.2.6. If so, then by Lemma 7.2.2 ($\boldsymbol{v}$ ($\boldsymbol{x}$ resp.) is a subsequence of $\boldsymbol{x}$ ($\boldsymbol{v}$ resp.)) we return $\boldsymbol{v} \prec \boldsymbol{x}$ ($\boldsymbol{x} \prec \boldsymbol{v}$ resp.), which decides that $\boldsymbol{p} \prec \boldsymbol{q}$ ($\boldsymbol{q} \prec \boldsymbol{p}$ resp.) and the algorithm terminates. Otherwise, we return to Step 2 with the inner context $\mathcal{C}^1\langle(\boldsymbol{v}_h, \boldsymbol{v}_{h^1}), (\boldsymbol{x}_h, \boldsymbol{x}_{h^1})\rangle$, where $h^1$ is the least integer such that $\boldsymbol{v}_{h^1} \neq \boldsymbol{x}_{h^1}$. In other words, we assign $\mathcal{C} = \mathcal{C}^1$ and then repeat Step 2 again.

To prove the correctness of the algorithm we need the following lemmas.

**Lemma 7.2.9** *Step 2 of Algorithm* `CompareV` *can be realized through a loop that considers each character $\alpha \in \Sigma$ in decreasing order, skipping the ones that are absent in both $\boldsymbol{v}$ and $\boldsymbol{x}$ or in the current context.*

*Proof.* In Step 2, we basically iterate from one (outer) context (say $\mathcal{C}_i$) to the next (inner) context (say, $\mathcal{C}_{i+1}$). Note that the substrings of $\mathcal{C}_i$ become the superstrings of $\mathcal{C}_{i+1}$. Recall that the substrings of a context are defined with respect to the $V$-forms of the superstrings. Now, by the definition of $V$-form, the largest letter in the superstrings cannot be present in the substrings. Hence, if we work with $\alpha_i$ and $\alpha_{i+1}$ while we are considering $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ respectively, then we must have $\alpha_i > \alpha_{i+1}$. Hence the result follows. □

**Lemma 7.2.10** *Algorithm* `CompareV` *terminates.*

*Proof.* Note that Algorithm `CompareV` can terminate only by conditions [C1] and [C2] of Step 2.a.(ii). Also recall that the input of the algorithm is two distinct strings. Furthermore, we have computed a global mismatch position $\ell_f$. Hence, clearly at some point we will reach either [C1] or [C2] of Step 2.a.(ii). Therefore, the algorithm will definitely terminate. □

The correctness of the algorithm follows immediately from Lemmas 7.2.1, 7.2.2, 7.2.9 and 7.2.10. Finally we analyze the running time of Algorithm `CompareV` as follows.

**Lemma 7.2.11** *Algorithm* `CompareV` *runs in $\mathcal{O}(n + |\Sigma|)$ time and space.*

*Proof.* The preprocessing in Step 1 of Algorithm `CompareV` builds the initial map in $\mathcal{O}(n+|\Sigma|)$ time and space (Lemma 7.2.4), and then identifies the first global mismatch. So, clearly, the preprocessing requires $\mathcal{O}(n + |\Sigma|)$ time and space.

The main algorithm revolves around Step 2, efficiently implemented as follows. Recall that, throughout this step we assume that the current context is $\mathcal{C}\langle(\boldsymbol{v}, \boldsymbol{v}_h), (\boldsymbol{x}, \boldsymbol{x}_h)\rangle$,

where $h$ is the least integer such that $\boldsymbol{v}_h \neq \boldsymbol{x}_h$. We consider all $\alpha \in \Sigma$ in decreasing order (Lemma 7.2.9). If a particular letter is absent in both the superstrings of the current context, we proceed to the next letter. So suppose we are currently considering $\alpha \in \Sigma$. Checking whether $\alpha$ is absent in both the superstrings $\boldsymbol{v}, \boldsymbol{x}$ of the current context can be done by traversing the two lists $\texttt{Map}_p(\alpha)$ and $\texttt{Map}_q(\alpha)$ in tandem. Note that we will always traverse the maps of the strings $\boldsymbol{p}$ and $\boldsymbol{q}$, although we may be considering different strings, $\boldsymbol{v}$ and $\boldsymbol{x}$, while we are within a particular context during the iterations of the algorithm. This works perfectly because, always, $\boldsymbol{v}$ and $\boldsymbol{x}$ would have to be substrings of $\boldsymbol{p}$ and $\boldsymbol{q}$: all we need is to keep track of the positions (range) of $\boldsymbol{v}$ and $\boldsymbol{x}$ in $\boldsymbol{p}$ and $\boldsymbol{q}$, i.e., some simple book-keeping information suffices to serve our purpose.

Once it is found that $\alpha$ is present in at least one of the superstrings, some work is necessary; note that most of this work is also simple book-keeping in order to map the positions in substrings to positions in the superstrings and vice versa. Undoubtedly, the most important task is to identify $h$. However, by Lemmas 7.2.5, 7.2.7 and Corollary 7.2.8, this index $h$ basically coincides with the global mismatch position $\ell_f$, and hence can also be identified easily through some simple book-keeping.

The algorithm terminates after traversing the two lists $\texttt{Map}_p(\alpha)$ and $\texttt{Map}_q(\alpha)$ in tandem. So, in total, the time required to complete this traversal and the tasks in Step 2 is $\mathcal{O}(n)$. Hence the result follows. □

We illustrate the algorithmic concepts with the following example.

**Example 7.2.12** *Let* $\boldsymbol{v} = 712734576$ *and* $\boldsymbol{x} = 71275174$. *With respect to their* $V$*-forms:* $\mathcal{L}_v = \mathcal{L}_x = 7$ *and* $j = k = 3$*; therefore* $Cond\text{-}I(\boldsymbol{v}, \boldsymbol{x})$ *holds. We have* $v_0 = x_0 = \epsilon$*;* $v_1 = x_1 = 12$*;* $v_2 = 345 \neq 51 = x_2$*, and hence* $h = 2$.

*The outer context is* $\mathcal{C}^0\langle(\boldsymbol{v}, \boldsymbol{v_2}), (\boldsymbol{x}, \boldsymbol{x_2})\rangle$ *and the inner context relates to the distinct substrings* $345$ *and* $51$ *giving* $\mathcal{C}'\langle(\boldsymbol{v_2}, \boldsymbol{v_0}), (\boldsymbol{x_2}, \boldsymbol{x_0})\rangle$*, with* $\mathcal{L}_{v_2} = \mathcal{L}_{x_2} = 5$ *and* $v_0 = 34 \neq \epsilon = x_0$.

*The algorithm preprocessing step computes the maps and mismatch position as follows.*

$\texttt{Map}_v(\Sigma)$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 2 | 3 | 5 | 6 | 7 | 9 | 1 |
| | | | | | | ↓ |
| | | | | | | 4 |
| | | | | | | ↓ |
| | | | | | | 8 |

$\texttt{Map}_x(\Sigma)$

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| 2 | 3 | | 8 | 5 | | 1 |
| ↓ | | | | | | ↓ |
| 6 | | | | | | 4 |
| | | | | | | ↓ |
| | | | | | | 7 |

*The global mismatch position is* $\ell_f = 5$ *and* $\alpha = \mathcal{L}_v = \mathcal{L}_x = 7$. *Proceeding to Step 2. the current context is* $\mathcal{C}\langle(\boldsymbol{v}, \boldsymbol{v_2}), (\boldsymbol{x}, \boldsymbol{x_2})\rangle$. *Step 2.a yields the sub-maps:*

$\texttt{Map}_{v_2}(7)$

| 3 | 4 | 5 |
|---|---|---|
| ↓ | ↓ | ↓ |
| 5 | 6 | 7 |

$\texttt{Map}_{x_2}(7)$

| 1 | 5 |
|---|---|
| ↓ | ↓ |
| 6 | 5 |

*For the inner context* $\mathcal{C}'\langle(\boldsymbol{v_2}, \boldsymbol{v_0}), (\boldsymbol{x_2}, \boldsymbol{x_0})\rangle$ *we have* $\beta = 5$ *and identify* $v_0 = 34 \neq \epsilon = x_0$. *Iterating, we find* $\texttt{Map}_{v_{h^2}}(4) = 6 \neq \texttt{Map}_{x_{h^2}}(4) = \emptyset$, *and hence we conclude that* $x \prec v$.

# Chapter III

# Improved solutions for molecular biology

# Article: # 1

# SimpLiSMS: A Simple, Lightweight and Fast Approach for Structured Motifs Searching

A Structured Motif refers to a sequence of simple motifs with distance constraints. Searching for such motifs is important, among others, in the context of identifying conserved features in biological sequences. We present SimpLiSMS [1] , a simple, lightweight and fast algorithm for searching structured motifs. We introduce the concept of a search context, character with respect to its preceding character according to the distance constraints of the structured motif. Our experiments show excellent performance of SimpLiSMS. Furthermore, we introduce a parallel version of SimpLiSMS which runs even faster. SimpLiSMS does not use any sophisticated data structure, which makes it simple and lightweight. And we believe it would be extremely useful for searching structured motifs in different contexts.

---

[1] Availability: SimpLiSMS is freely available for use by anyone (including the source codes) at the following address: http://www.ekngine.com/SimpLiSMS/.

## 1.1 Introduction

A *Structured Motif* (alternatively, a *structured pattern*) is defined by a list of simple (as opposed to structured) sub-patterns (or seeds) separated by variable length gaps defined by a list of intervals [MPVZ05, CS04]. In other words, a structured motif imposes a sort of variable constraint on the relative distances among its sub-patterns: between two consecutive sub-patterns, a structured motif allows any gap within the minimum and maximum limit provided as part of the definition. It is also referred to as "compound patterns" in [MPVZ05, CS04]. More formally, a structured motif can be defined as a pair $(\mathcal{S}, \mathcal{G})$, where $\mathcal{S} = (S_1, ..., S_k)$ is a sequence of seeds (i.e., patterns) and $\mathcal{G} = ([a_1, b_1], ..., [a_{k-1}, b_{k-1}])$, with $a_i, b_i \in \mathbb{Z}$ and $a_i \leq b_i$ for $1 \leq i < k$ is a sequence of closed intervals characterizing the gaps between the consecutive seeds.

Structured motifs find interesting and useful applications in computational biology and bioinformatics. For example, the PROSITE database [HBB+06, SdCC+13] supports searching for structured motifs. Different application scenarios for structured motif pop up during different biological experiments. This is especially useful in the identification of conserved features in a set of DNA or protein sequences.

To motivate the readers, the structured motif of the form

$$MT[115, 136]MTNTAYGG[121, 151]GTNGAYGAY$$

reported in [MPVZ05] may be cited. In particular, the authors in [MPVZ05] refer to a biological experiment, where the exact goal was to localize several LTR retrotransposons [2] and to establish, by multiple alignment, a number of conserved features. And they report that the experimental data from about 10% of the rice genome identified the above-mentioned structured motif in many retrotransposons belonging to the Ty1-copia group in correspondence to the gene encoding the reverse transcriptase [3]. The structured motif mentioned above consists of three seeds (sub-patterns), namely, $MT$, $MTNTAYGG$ and $GTNGAYGAY$ written in IU-PAC alphabet[4], and two intervals

[2] Retrotransposons (also called transposons via RNAintermediates) are genetic elements that can amplify themselves in a genome and are ubiquitous components of the DNA of many eukaryotic organisms.

[3] Reverse transcriptase, also called RNA-directed DNA polymerase, an enzyme encoded from the genetic material of retroviruses that catalyzes the transcription of retrovirus RNA (ribonucleic acid) into DNA (deoxyribonucleic acid).

[4] IU-PAC alphabet will be discussed later in the context of a degenerate string. The reader may refer

imposing constraints on the relative distances between adjacent patterns. In particular, this means that the gap between the first and second (second and third) seeds is greater than 114 bps (120) but less than 137 bps (152).

As another motivation, we cite the application and usefulness of structured motif search in the context of protein matching as mentioned in [NR03, NR01]. Consider the PROSITE database [HBB+06, SdCC+13] which contains protein site descriptions. For each protein site, the database contains an expression containing character classes (i.e., more than once character in a position) and bounded sized gaps. Clearly, this expression is essentially a structured motif. To elaborate, we borrow the example cited in [NR03]. Consider the expression of the protein site number PS00007: $[RK][2,3][DE][2,3]Y$. This expression gives us a structured motif comprising three sub-patterns: the first sub-pattern is a single character which could be either $R$ (i.e., the amino acid *Arginine*) or $K$ (i.e., the amino acid *Lysine*), the second one is also a single character which could be either $D$ (i.e., the amino acid *Aspartic acid*) or $E$ (i.e., the amino acid *Glutamic acid*), and the third and last one is another single character $Y$ (i.e., the amino acid *Tyrosine*). And from the distance constraints it is clear that the gap between the first and second (second and third) sub-patterns is greater than 2 but less than 3.

We note here that the concept of a structured motif is somewhat different from a related concept of *"metamotifs"* or *"motifs of motifs"*, which refers to an expression that specifies a particular arrangement of motifs [GBEB97, JPB01, BBB+09, BWML06]. Also, note that, in the literature the terms *'motif'* and *'pattern'* are sometimes used interchangeably and sometimes for slightly different meanings. For example, *pattern searching* usually refers to the problem of locating a given pattern in one or more given texts. On the the other hand, *motif searching* in many contexts refers to the problem of identifying patterns that are found frequently (and possibly with some more specific properties) in the given text(s). Note that in the latter case, the pattern is not part of the input whereas in the former it is. To this end, the problem we handle belongs to the former group of problems.

The problem of structured motif search has received significant attention in the literature. The simplest approach is to solve this problem using a regular expression matching (REM) algorithm. But as has been argued by [BGVW12], such approach

to Columns 1, 2 and 3 of Table 1.1 at this point.

cannot be efficient unless some special care is taken in the translation of the problem to REM. Among theoretically efficient algorithms for this problem using REM, the recent work by [BT10] is notable. [NR03, NR01] presented a very fast and practical implementation of an REM algorithm to solve the problem exploiting bit parallelism. However, there algorithm becomes less efficient as gaps get longer [BGVW12]. Also, bit operations are more costly when they have to be performed on several computer words instead of one. An alternative approach, suggested independently by [MPVZ05] and [RIL+06], is to design algorithms in two phases. In the first phase, the occurrences of the seeds of the structured motif are computed and in the second, the intervals are considered to identify the occurrences of the complete structured motif. In what follows, the algorithms handling this problem using the 2-phase approach will be referred to as 2-$\phi$-algorithms. Additionally, [MPVZ05] performed extensive experiments to examine the usefulness of this approach. In the implementation of the algorithm of [MPVZ05], suffix trees were used in the first phase to compute the occurrences of the seeds. One of the problem of this approach is the huge space requirement due to the summation of the number of occurrences of all the seeds, many of which ultimately may turn out to be useless in the context of the actual occurrences of the complete structured motif. Apart from the above we are aware of two more works namely, SMOTIF [ZZ06] and a follow up work on SMOTIF in [HS08] as a conference paper. Notably the work of [HS08] almost resembles the 2-$\phi$-algorithms mentioned above and it uses suffix tree as the index and hence suffers the same problem suffered by [MPVZ05] as mentioned above. Unfortunately most of the prior works including [MPVZ05, RIL+06] were not cited in [HS08].

In this article, we present SimpLiSMS (pronounced "Simply SMS"), a simple, lightweight and fast algorithm for searching structured motifs. SimpLiSMS exploits an idea of a search context (to be defined, shortly) and combines the two phases of 2-$\phi$-algorithms into one. It identifies the occurrences of the seeds, mostly through a character by character matching, rather than a seed by seed matching and thereby refrains from using a heavy-weight data structure like a suffix tree. As a result, not only that SimpLiSMS is lightweight, as it turns out, it is also extremely fast in practice. Moreover, SimLiSMS lends itself easily to a parallel implementation which makes the searching even faster.

## 1.2 Preliminaries

In this work, unless otherwise specified, the underlying alphabet is assumed to be the DNA alphabet, i.e., $\Sigma = \{A, C, G, T\}$.

**Definition 36** *A structured motif can be defined as a pair* $(\mathcal{S}, \mathcal{G})$*, where* $\mathcal{S} = (\boldsymbol{s_1}, .., \boldsymbol{s_k})$ *is a sequence of seeds (i.e., patterns) and* $\mathcal{G} = ([a_1, b_1], .., [a_{k-1}, b_{k-1}])$*, with* $a_i, b_i \in \mathbb{Z}$ *and* $a_i \leq b_i$ *for* $1 \leq i < k$ *is a sequence of closed intervals characterizing the gaps between the consecutive seeds. So, in an occurrence of a structured motif, the distance between two consecutive seeds* $\boldsymbol{s}_{k-1}$ *and* $\boldsymbol{s}_k$ *must be within the close interval* $[a_{k-1}, b_{k-1}]$*. A structured motif* $\mathcal{M}$ *is usually expressed in the following form:*

$$\mathcal{M} = \boldsymbol{s_1}\ [a_1, b_1]\ \boldsymbol{s_2}\ [a_2, b_2]\ \ldots\ \boldsymbol{s_{k-1}}\ [a_{k-1}, b_{k-1}]\ \boldsymbol{s}_k. \tag{1.1}$$

**Problem 1.2.1** *(Structured Motif Search)*
*We are given a sequence* $\boldsymbol{x}$ *and a structured motif* $\mathcal{M} = (\mathcal{S}, \mathcal{G})$*. We need to find the occurrences of* $\mathcal{M}$ *in* $\boldsymbol{x}$*.*

We exploit the idea of a search context which is defined as follows.

**Definition 37** *(Search Context). Given a sequence* $\boldsymbol{x}$*, a search context is the smallest factor of* $\boldsymbol{x}$ *that starts with an occurrence of the first seed* $\boldsymbol{s}_1$ *and has the maximum length equal to* $\sum_{i=1}^{k-1}(|\boldsymbol{s}_i| + b_i) + |\boldsymbol{s}_k|$*.*

We use the following notations, with respect to a search context. We use $\mathcal{L}_{min}(\mathcal{L}_{max})$ to denote the minimum (maximum) possible length of a search context. More formally, we have the following:

$$\mathcal{L}_{min} = \sum_{k=1}^{k=i-1} |\boldsymbol{s}_i| + a_i + |\boldsymbol{s}_k|$$

$$\mathcal{L}_{max} = \sum_{k=1}^{k=i-1} |\boldsymbol{s}_i| + b_i + |\boldsymbol{s}_k|.$$

We maintain a data structure called $\mathcal{M}ap$ which is defined as follows.

**Definition 38** *($\mathcal{M}ap$). Given a structured motif $\mathcal{M}$, assume that $\boldsymbol{y} = \boldsymbol{s}_1\boldsymbol{s}_2 \ldots \boldsymbol{s}_k$ and let $|\boldsymbol{y}| = \ell$. Then $\mathcal{M}ap[1 \ldots \ell]$ is an array of length $\ell$ where $\mathcal{M}ap[i], 1 \leq i \leq \ell$ is the pair $(A, B)$ as defined below:*

$$\mathcal{M}ap[i].A = \begin{cases} i; & \text{if } i = 0 \\ \mathcal{M}ap[i-1].A + a_{k'} + 1; & \text{if } i = \sum_{j=1}^{k'} |\boldsymbol{s}_j|, \text{for some } k' < k. \\ \mathcal{M}ap[i-1].A + 1; & \text{otherwise} \end{cases}$$

$$\mathcal{M}ap[i].B = \begin{cases} i; & \text{if } i = 0 \\ \mathcal{M}ap[i-1].B + b_{k'} + 1; & \text{if } i = \sum_{j=1}^{k'} |\boldsymbol{s}_j|, \text{for some } k' < k. \\ \mathcal{M}ap[i-1].B + 1; & \text{otherwise} \end{cases}$$

In other words, given a position $i$, the range $[\mathcal{M}ap[i].A, \mathcal{M}ap[i].B]$ gives us the valid positions for $\mathcal{M}[i]$. Example 1.2.2 computes the $\mathcal{M}ap$ for $\mathcal{M} = CATA[1,3]TACA[0,2]GGG$.

**Example 1.2.2** *Given the pattern (structured motif)* $\mathcal{M}$=$CATA[1,3]TACA[0,2]GGG$, *Algorithm ConstructMap (Figure 1.12) our algorithm will construct the $\mathcal{M}ap$ as follows:*

$$\mathcal{M} = CATA[1,3]TACA[0,2]GGG$$

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{y}$ | $C$ | $A$ | $T$ | $A$ | $T$ | $A$ | $C$ | $A$ | $G$ | $G$ | $G$ |
| $\mathcal{M}ap$ | [0,0] | [1,1] | [2,2] | [3,3] | [5,7] | [6,8] | [7,9] | [8,10] | [9,13] | [10,14] | [11,15] |

## 1.3 Methods

As has been mentioned by [RIL+06], there could be an explosive number of occurrences of a structured motif especially because different occurrences of it can exist having exactly the same start and end positions. This happens because of the so called 'elasticity' of the gaps, i.e., variable length gaps. To avoid reporting such explosive number of occurrences we exploit the concept of a search context. Given a sequence $\boldsymbol{x}$, a search context is the smallest factor of $\boldsymbol{x}$ that starts with an occurrence of the first

seed $\boldsymbol{s}_1$ and has length equal to $\sum_{i=1}^{k-1}(|\boldsymbol{s}_i| + b_i) + |\boldsymbol{s}_k|$. In other words, the longest possible pattern corresponding to the structured motif can (barely) fit in the search context.

SimpLiSMS takes a simple approach. It identifies all possible search contexts in $\boldsymbol{x}$ and consider each of those one after another. In each search context it checks whether there is indeed an occurrence of the structured motif in it and if yes, it identifies and report the start position of that structured motif. Notably, the start position of the search context is the start position of the structured motif that exists in it. Then it moves to the next search context and so on. Since a search context is defined by the occurrence of the first seed, SimpLiSMS uses an exact pattern matching algorithm (e.g., the famous KMP algorithm for exact matching [KMP77]) to identify all the occurrences of the first seed and thereby identify all the search contexts. To facilitate the search process within a search context, it makes use of a data structure called $\mathcal{M}ap$ that keeps track of the valid positions for a particular character in the structured motif with respect to its preceding character. $\mathcal{M}ap$ is constructed as a preprocessing step before the actual search in a search context can begin.

A parallel version of the SimpLiSMS, referred to as SimpLiSMS-P, is also implemented as follows. The sequence $\boldsymbol{x}$ is first divided into $f$ overlapping subsequences $\{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_f\}$ of length $\ell$ each, except possibly for the last one, $\boldsymbol{x}_f$, which may have a lesser length. The overlapping is done so that no search context is missed due to the cutting of the sequence into subsequences. Each subsequence $\boldsymbol{x}_i, 1 \leq i \leq f$ is then handled as a separate thread or process in a multi-processor/multi-threaded machine.

## 1.4 SimpLiSMS Algorithm outline

Recall that the input of our problem is a sequence $\boldsymbol{x}$ and a structured motif $\mathcal{M}$. An outline of the $SimpLiSMS$ algorithm is as follows.

1. **PREPROCESSING PHASE**

   **Step 1. [Preprocessing the sequence $\boldsymbol{x}$]:** SimpLiSMS segments the sequence $\boldsymbol{x}$ into overlapping factors of length $\ell$, called $\ell$-factors, where $\ell > \mathcal{L}_{max}$, such that each consecutive $\ell$-factors overlap with each other by $\mathcal{L}_{max}$. The overlap is to cover all possible search contexts in $\boldsymbol{x}$.

**Step 2. [Preprocessing the pattern $\mathcal{M}$]:** In this step, SimpLiSMS constructs $\mathcal{M}ap$ Algorithm ConstructMap (Figure 1.12) for the structured motif $\mathcal{M}$.

**Step 3. [Preprocessing for exact string matching]:** SimpLiSMS uses an exact string matching algorithm, e.g., the KMP algorithm, to identify the start positions of the first seed $\boldsymbol{s}_1$. For the KMP algorithm, it needs to compute the so called failure function table $\pi$ (Figure 1.15) for the first seed $\boldsymbol{s}_1$ of the structured motif $\mathcal{M}$. We have also implemented SimpLiSMS with Boyer-Moore (BM) algorithm [BM77] (Figure 1.20). For BM algorithm, it needs to build the *bad character shift array* (Figure 1.17) and *good suffix shift array* (Figures 1.19 and 1.18). Such preprocessing is done during this step.

2. **PATTERN MATCHING PHASE**

**Step 1.** The algorithm searches the $\ell$-factors obtained in Step 1.1. For each $\ell$-factor the algorithm finds $\alpha$, the list of starting positions of all occurrences of the first seed $\boldsymbol{s}_1$ of the structured motif $\mathcal{M}$ within the given $\ell$-factor Algorithm KMPSearch (Figure 1.16) and Algorithm BoyerMooreSearch (Figure 1.20).

**Step 2.** For each match of $\boldsymbol{s}_1$ we calculate the boundaries of the search context ($\mathcal{L}_{min}$ and $\mathcal{L}_{max}$) relative to the position of $\boldsymbol{s}_1$ in the sequence $\boldsymbol{x}$.

**Step 3.** Now SimpLiSMS determines whether there exist at least one occurrence of the structured motif $\mathcal{M}$ in the current search context. The algorithm (Figure 1.14) performs a guided search for $\mathcal{M}$ in the current search context $SC$ with the help of $\mathcal{M}ap$ computed during the pre-processing. Suppose the start position of the current search context is $p$. Recall that, $\boldsymbol{y} = \boldsymbol{s}_1\boldsymbol{s}_2 \ldots \boldsymbol{s}_k$ and the occurrence of $\boldsymbol{s}_1$ coincides with the start of the search context. So, SimpLiSMS starts checking for a valid match from $\boldsymbol{y}[|\boldsymbol{s}_1| + 1]$. Now, suppose we have valid matches up to $\boldsymbol{x}[i_1] = \boldsymbol{y}[j], j > |\boldsymbol{s}_1|$. Now we are going to check $\boldsymbol{y}[j + 1]$. Suppose we have $\boldsymbol{x}[i_2] = \boldsymbol{y}[j + 1]$. Then, SimpLiSMS only needs to check whether $i_2 - p + 1 \in [\mathcal{M}ap[i_2].A, \mathcal{M}ap[i_2].B]$. If yes, then we continue to check $\boldsymbol{y}[j+1]$. Otherwise, we need to start re-checking from $\boldsymbol{y}[|\boldsymbol{s}_1| + 1]$ all over again.

**Step 4.** If $\mathcal{M}$ is found in the current searching context, then SimpLiSMS reports

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\boldsymbol{x}[i]$ | $a$ | $a$ | $[abc]$ | $a$ | $[ac]$ | $b$ | $c$ | $a$ | $a$ | $[ac]$ | $b$ | $a$ | $c$ | $[abc]$ | $a$ | $[bc]$ |

Figure 1.1: An example of a degenerate string.

the start position of the search context and proceed to the next search context.

## 1.5 Handling Degenerate Strings

A degenerate string (also referred to as the indeterminate string in the literature) is a sequence $\boldsymbol{x} = \boldsymbol{x}[1\,..\,n]$, where $\boldsymbol{x}[i] \subseteq \Sigma$ for all $i$. A position of a degenerate string may match more than one elements from the alphabet $\Sigma$; such a position is said to have a *non-solid* symbol (also called a character class). If in a position we have only one element of $\Sigma$, then we refer to this position as *solid*. The length of a degenerate string is defined in the same way as it is for a regular string: a degenerate string $\boldsymbol{x}$ has length $n$, when $\boldsymbol{x}$ has $n$ positions, where each position can be either solid or non-solid. We represent non-solid positions using $[..]$ and solid positions omitting $[..]$. The example in Figure 1.1 identifies the solid and non-solid positions of a degenerate string.

For degenerate strings the definition of a matching relation is extended as follows. A degenerate character (or character class) $s_1$ is said to match another degenerate character $s_2$, if and only if $s_1 \bigcap s_2 \neq \epsilon$.

**Problem 1.5.1** *(Degenerate Structured Motif Search)*
*Given a degenerate sequence $\boldsymbol{x}$ and a structured motif $\mathcal{M}$, compute all starting positions of $\mathcal{M}$ in $\boldsymbol{x}$.*

Note that there could be $2^\sigma - 1$ possible subsets of $\Sigma$ and hence there are in total as many degenerate characters including the $\sigma$ solid characters. For example, for DNA alphabet there are 4 letters, namely $A, C, G$ and $T$. Hence there could be 15 degenerate characters including the 4 solid characters. In order to efficiently match degenerate characters, we represent each degenerate character as a sequence of $|\Sigma|$ bits. We maintain an array of bit masks $\mathcal{U}$ of length $2^\sigma - 1$ in a way such that the bit mask of any given degenerate symbol (solid or non-solid) can be accessed in constant time.

For each character of the DNA alphabet, i.e., $\{A, C, G, T\}$ the conversion to the corresponding 4 bit mask is done as follows:

- $\mathcal{U}(A) = 0001$
- $\mathcal{U}(C) = 0010$
- $\mathcal{U}(G) = 0100$
- $\mathcal{U}(T) = 1000$

Then, the bit mask of each non-solid symbol $s$ can be computed as follows. Suppose $s$ contains the characters $a_{i_1}, a_{i_2}, ..., a_{i_k}$ where $k \leq 4$ and $a_{i_j} \in \Sigma, 1 \leq j \leq k$. Then the bit mask of $\boldsymbol{s}$, namely, $\mathcal{U}(\boldsymbol{s}) = \mathcal{U}(a_{i_1})$ OR $\mathcal{U}(a_{i_2})$ OR$\ldots$ OR $\mathcal{U}(a_{1_k})$. Clearly, given two degenerate characters $s_1$ and $s_2$, now we can say that $s_1$ and $s_2$ (degenerate) match if and only if $\mathcal{U}(s_1)$ AND $\mathcal{U}(s_2) > 0$. With these bit masks defined for each (solid/non-solid) character, we can easily check whether two degenerate characters match using the following easy lemma.

**Lemma 1.5.2** *Given two degenerate characters $s_1$ and $s_2$, we say $s_1$ and $s_2$ (degenerate) match if and only if*

$$\mathcal{U}(s_1) \text{ AND } \mathcal{U}(s_2) > 0.$$

□

For example, to determine whether $[AC]$ matches with $[CD]$ (and vice versa), first we convert the symbols into corresponding bit masks as follows: $[AC] \equiv 0011$ and $[CG] \equiv 0110$. Then we perform AND operation on the bit masks as follows:

$$0011 \; AND \; 0110 = 0010 > 0$$

Since, we have a non-zero result, we can conclude that $[AC]$ matches with $[CD]$. Notably, the match is due to $C$ which is the common symbol between the two degenerate characters considered above.

Degenerate characters or character classes can be found in biological sequences and are ubiquitous in PROSITE database [SdCC+13, HBB+06]. This is why, search in

| Symbol | Description | Bases | Code |
|---|---|---|---|
| A | Adenine | A | 0001 |
| C | Cytosine | C | 0010 |
| G | Guanine | G | 0100 |
| T | Thymine | T | 1000 |
| R | puRine | A **or** G | 0101 |
| Y | pYrimidine | C or T | 1010 |
| K | Keto | G **or** T | 1100 |
| M | aMino | A **or** C | 0011 |
| S | Strong | C **or** G | 0110 |
| W | Weak | A **or** T | 1001 |
| B | not A | C **or** G **or** T | 1110 |
| D | not C | A **or** G **or** T | 1101 |
| H | not G | A **or** C **or** T | 1011 |
| V | not T | A **or** C **or** G | 0111 |
| N | aNy base | A **or** C **or** G **or** T | 1111 |

Table 1.1: IU-PAC Extended DNA Alphabet and corresponding bit masks.

PROSITE database supports the use of character classes. Notably, the IU-PAC alphabet is the extended DNA alphabet to capture this notion of degenerate characters (Table 1.1).

Now, SimpLiSMS can be extended for degenerate structured motif searching by simply plugging in the $isEquivelent$ function (Figure 1.13) for checking the degenerate matching as described above.

## 1.6 Results

We have evaluated the performance of SimpLiSMS through extensive experiments. We have used 4 sequences, namely, a sequence taken from the *Homo sapiens* genome (size: 256,053,182 bytes), the *Arabidopsis thaliana* DNA sequence (size: 321,118,972 bytes), the *Oryza sativa* DNA sequence (size: 634,849,961 bytes) and finally a randomly generated sequence (size: 104,8576,000 bytes). We have followed the experimental strategy of [MPVZ05]. A set of 1000 structured motifs were randomly generated by randomly choosing, for each one, the number $k \in [3, 8]$ of simple motifs, the length $\ell \in [5, 10]$ of each motif and $k - 1$ intervals of $[0, 100]$ as variable length gaps. The experiments were run on a Windows Server 2008 R2 64-bit Operat-

| Name | Searching Algorithm (for $S_1$) | Sequential/Parallel |
|---|---|---|
| SimpLiSMS-KMP-S | KMP | Sequential |
| SimpLiSMS-KMP-P | KMP | Parallel |
| SimpLiSMS-BM-S | Boyer-Moore | Sequential |
| SimpLiSMS-BM-P | Boyer-Moore | Parallel |

Table 1.2: Different Variants/Implementations of SimpLiSMS.

ing System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. We have implemented SimpLiSMS in $C\#$ language using Visual Studio 2010. To compare the performance we have also implemented the 2-$\phi$-algorithm of [RIL+06] and [MPVZ05] using the Aho-Corasick pattern matching machine of [AC75] to implement the first phase (i.e., search phase). Due to the huge space requirement of suffix tree we did not use the suffix tree in the search phase of the 2-$\phi$-algorithm. We also slightly modify Phase 1 of the 2-$\phi$-algorithm to incorporate the concept of a search context to ensure a level-playing ground with SimpLiSMS.

We do not compare SimpLiSMS with the work of [NR03, NR01] because the length of the structured motif we use in our experiments are larger than computer words, for which their algorithm is reported to be quite slow. This is why their algorithm was not considered in the experimentation of [MPVZ05] as well.

Although originally we devised SimpLiSMS using KMP algorithm, we also implemented a variant where KMP algorithm was replaced by the famous Boyer-Moore algorithm [BM77] (Figure 1.20). Our motivation for this comes from the fact that despite much better theoretical running time, the Boyer-Moore algorithm outperforms KMP algorithm in practice. And indeed as will be reported shortly, the performance of SimpLiSMS with Boyer-Moore algorithm performs better in most cases. Table 1.2 describes the different implementations of our algorithm.

The results are illustrated in Figures 1.2 through 1.11. These figures basically present two different types of comparisons. Since the size of the structured motif largely depends on the gap length, in Figures 1.2, 1.4, 1.6 and 1.10 the time required to compute the occurrences of the set of structured motifs are reported against the gap lengths in those. On the other hand, the number of occurrences also affect the search time significantly. Hence, in Figures 1.3, 1.5, 1.7 and 1.11, the time vs. number of occurrences are plotted. In particular, in Figures 1.2 and 1.3, we present the com-

parison among the 2-$\phi$-algorithm, SimpLiSMS-KMP-S and SimpLiSMS-KMP-P. On the other hand, in Figures 1.4 and 1.5, we present the comparison among the 2-$\phi$-algorithm, SimpLiSMS-BM-S and SimpLiSMS-BM-P. Finally, in Figures 1.6 and 1.7 we put SimpLiSMS-KMP-S and SimpLiSMS-BM-S against each other. From the figures, performance superiority of SimpLiSMS over the 2-$\phi$ algorithm is clearly evident. It is also clear that SimpLiSMS-P runs even faster as expected. Also, in most cases, SimpLiSMS-BM outperforms SimpLiSMS-KMP.

At this point a brief discussion on SimpLiSMS-P is in order. We note that SimPliSMS-P can be configured depending of the length of the input sequence and the machine resources (RAM size). For example to run a search on a sequence of length 600MB, we configure SimpLiSMS to segment the sequence into 75 smaller segments each of length 8MB. So the queue contains 75 threads and we set the concurrent thread limit to 25 (the number of threads running at anytime).

We have also made an attempt to compare SimpLiSMS with SMOTIF [ZZ06]. However the implementation available for SMOTIF turned out to be a bit problematic and was crashing for long motifs. As a result we could not run the experiments for longer gap length. However, as is evident from Figures 1.8 and 1.9, the run-time of SMOTIF remains almost invariant with regards to the changing gap length or the number of occurrences. And clearly, the performance of SimpLiSMS is better than SMOTIF.

We also have considered degenerate motifs. As a second experiment, we compared the performances of SimpLiSMS-BM, SimpLiSMS-KPM and the 2-$\phi$ algorithm by processing the same data-set used in the first experiment to search for a set of 1,000 degenerate structured motifs over the IUPAC alphabet, randomly generated by randomly choosing, for each model, the length $\ell \in [5, 10]$ of each motif and $k - 1$ intervals of $[0, 100]$ as variable length gaps. The results, averaged over 10 trials, are illustrated in Figures 1.10 and 1.11. As expected, the performance superiority of SimpLiSMS over 2-$\phi$ algorithm is clearly evident from the figures.

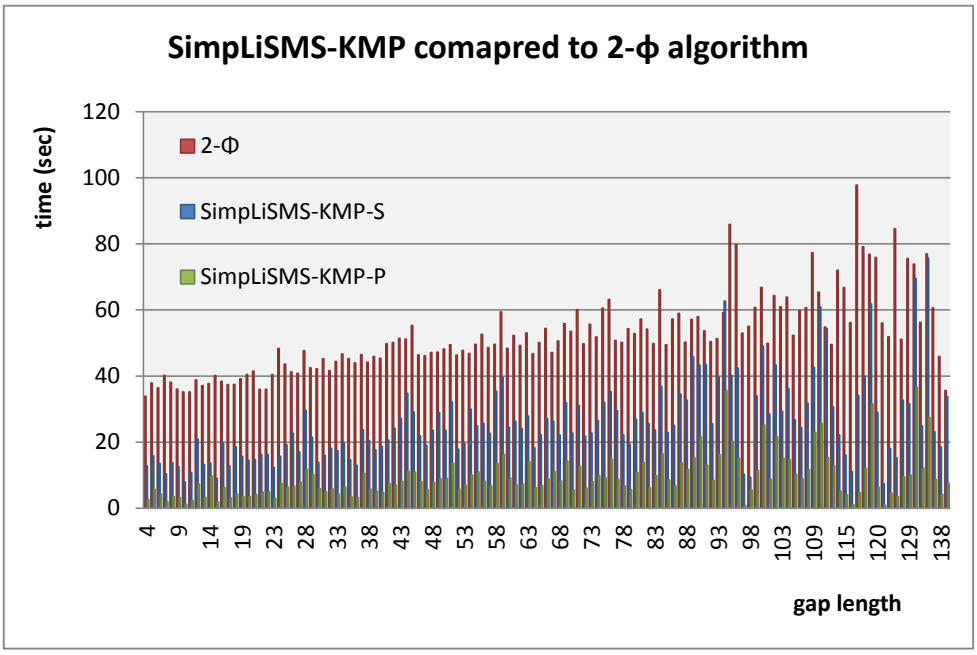



Figure 1.2: Comparison of SimpLiSMS-KMP-S, SimpLiSMS-KMP-P and 2-$\phi$-algorithm (time vs. gaps length).

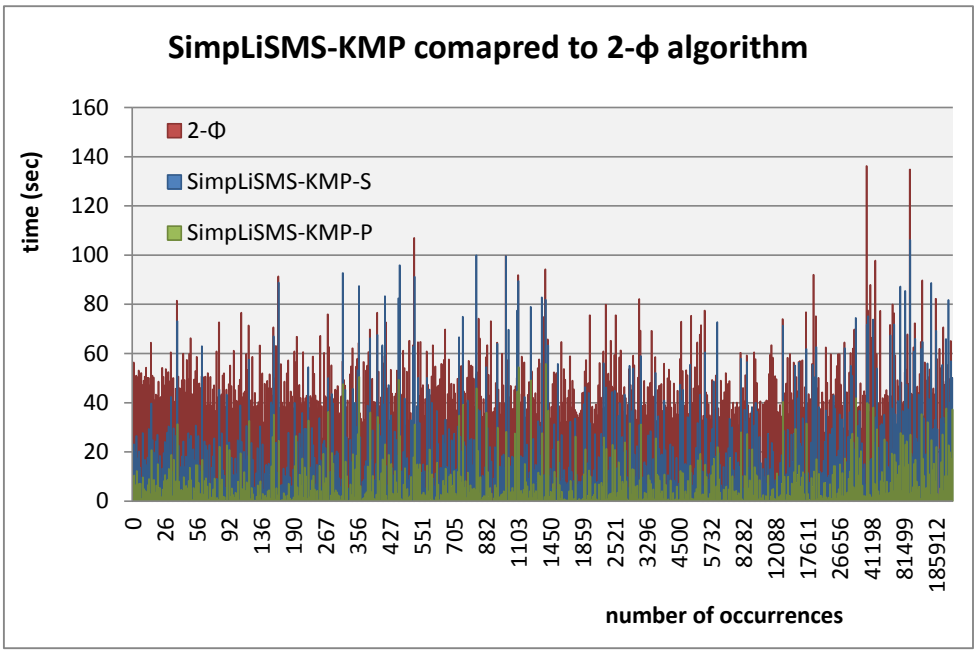



Figure 1.3: Comparison of SimpLiSMS-KMP-S, SimpLiSMS-KMP-P and 2-$\phi$-algorithm (time vs. number of occurrences).

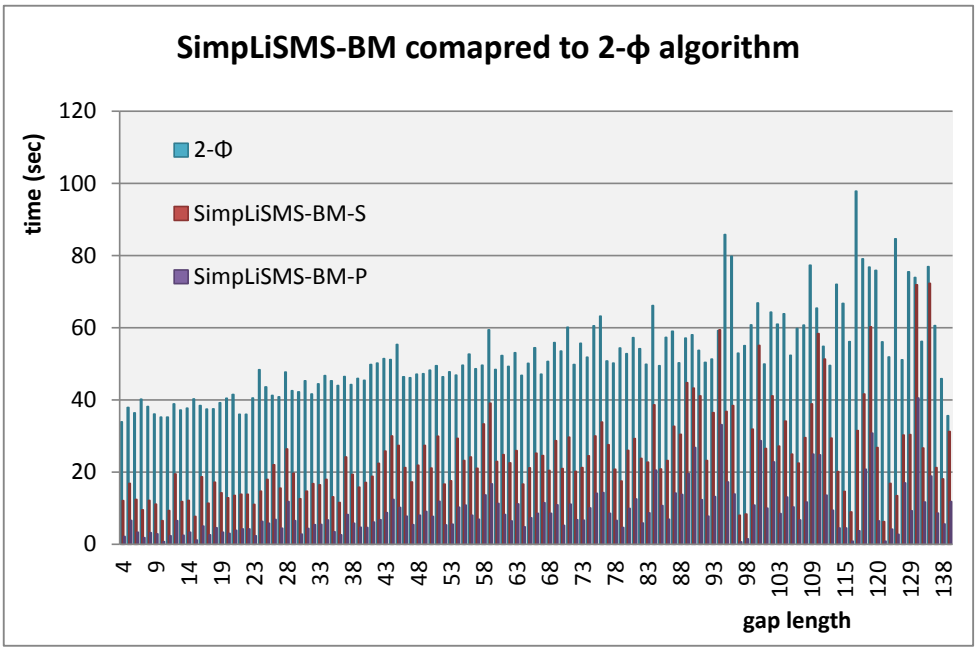



Figure 1.4: Comparison of SimpLiSMS-BM-S, SimpLiSMS-BM-P and 2-$\phi$-algorithm (time vs. gaps length).

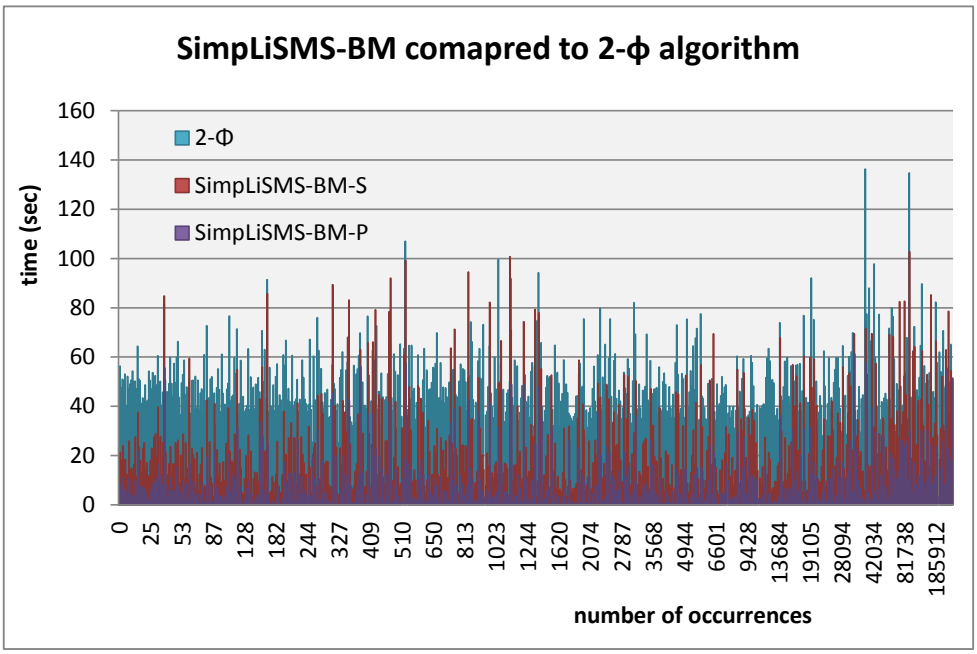



Figure 1.5: Comparison of SimpLiSMS-BM-S, SimpLiSMS-BM-P and 2-$\phi$-algorithm (time vs. number of occurrences).

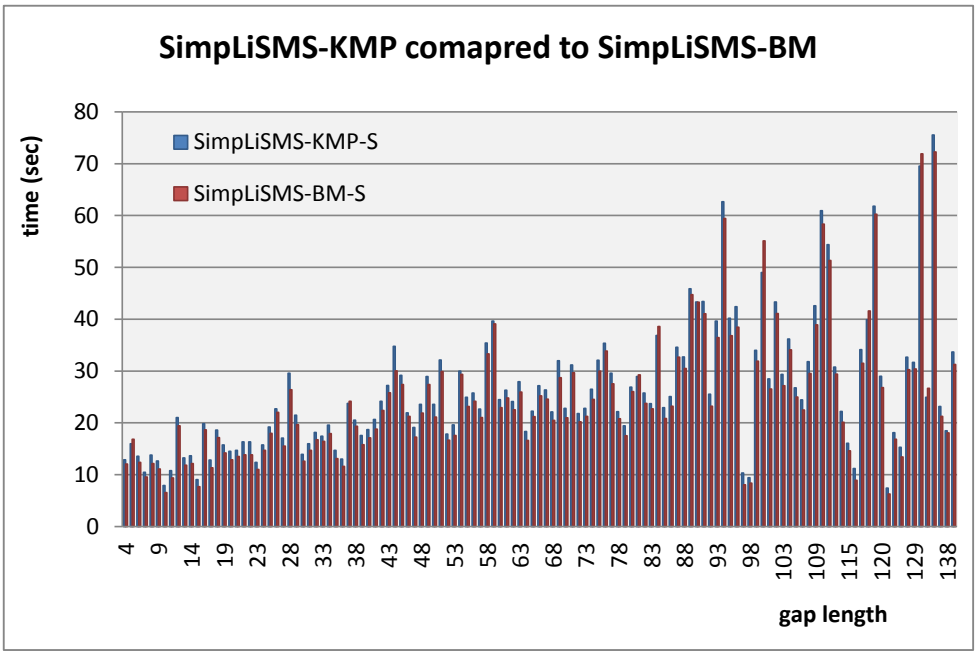



Figure 1.6: Comparison of SimpLiSMS-BM-S and SimpLiSMS-KMP-S (time vs. gaps length).

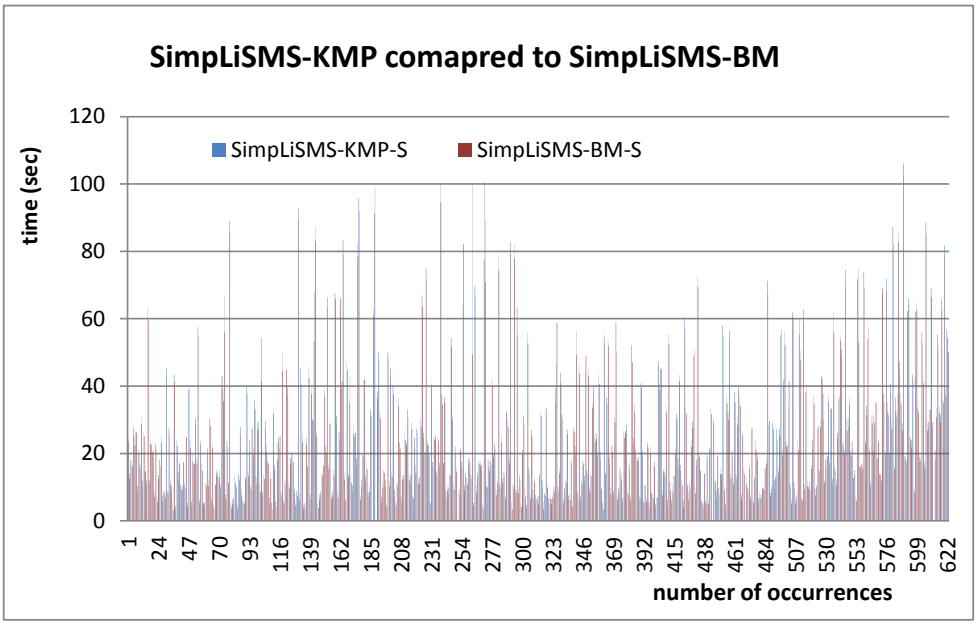



Figure 1.7: Comparison of SimpLiSMS-BM-S and SimpLiSMS-KMP-S (time vs. number of occurrences).

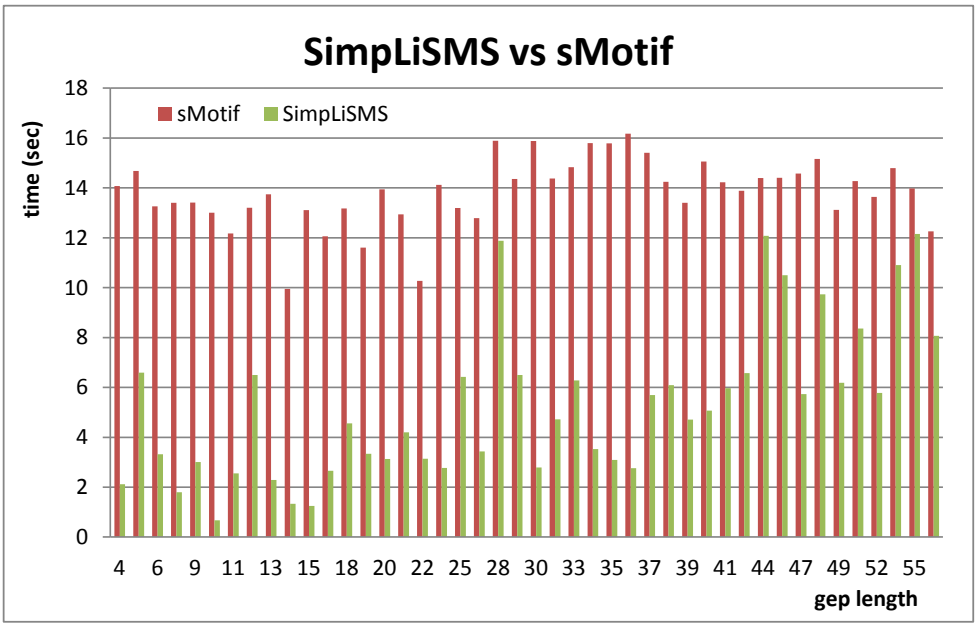



Figure 1.8: Comparison of SimpLiSMS and sMotif (time vs. gaps length).

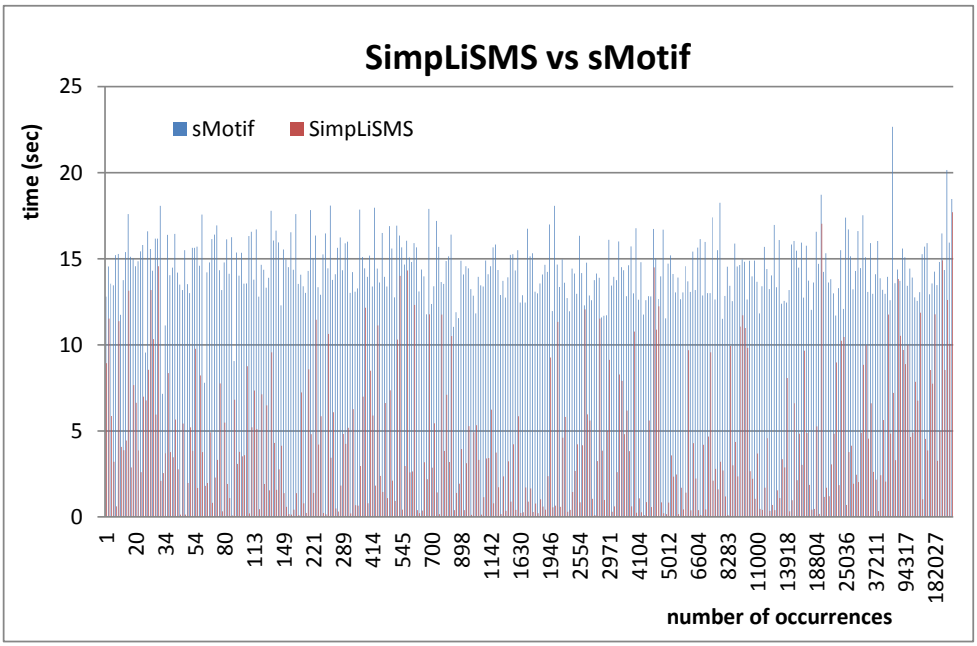



Figure 1.9: Comparison of SimpLiSMS and sMotif (time vs. number of occurrences).

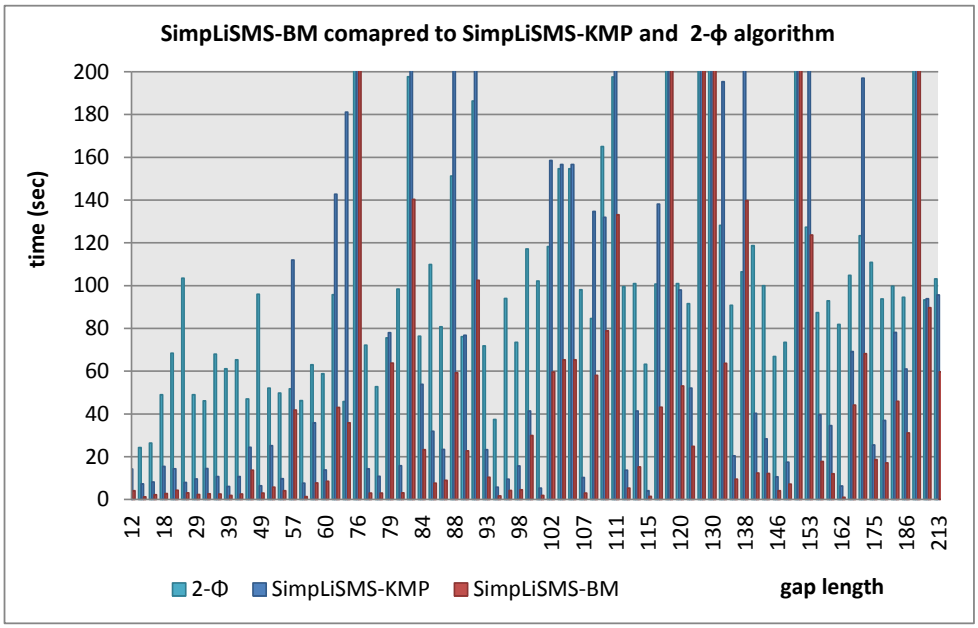



Figure 1.10: Comparison of SimpLiSMS-BM, SimpLiSMS-KMP and 2-$\phi$-algorithm for degenerate structured motifs (time vs. gaps length).

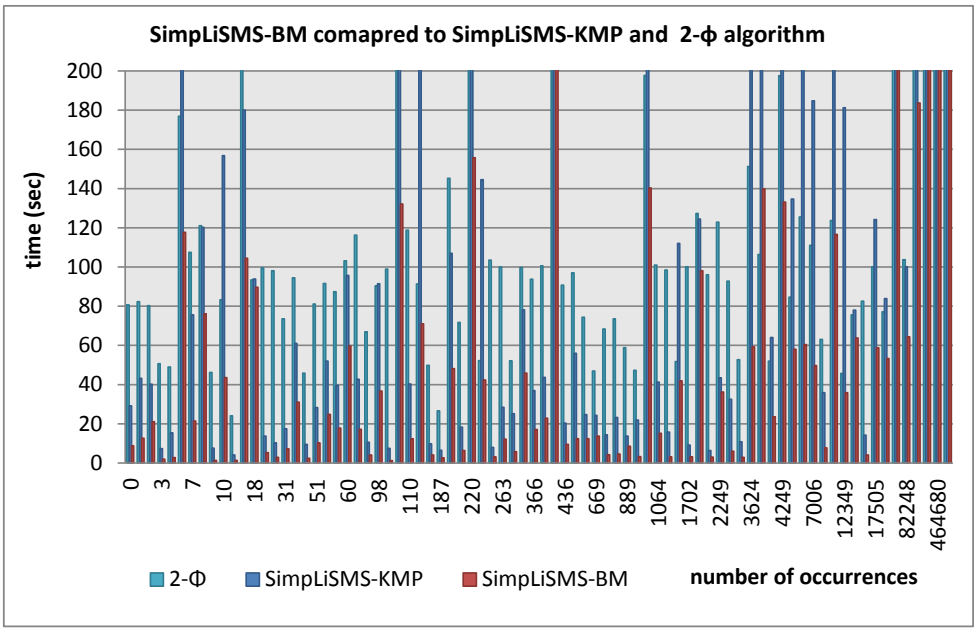



Figure 1.11: Comparison of SimpLiSMS-BM, SimpLiSMS-KMP and 2-$\phi$-algorithm for degenerate structured motifs (time vs. number of occurrences).

## 1.7 Pseudocode

**procedure** CONSTRUCTMAP($(\mathcal{S}, \mathcal{G})$)
  $\mathcal{M}ap[1..n, 1..2] \leftarrow 0^{n\times 2}$; $index \leftarrow 0$; $\mathcal{L}_{min}, \mathcal{L}_{max} \leftarrow 0$
  **for** $(i \leftarrow 1$ **to** $|\mathcal{S}|)$ **do**
    $\mathcal{L}_{max} = \mathcal{L}_{max} + |S_{i-1}| + \mathcal{G}[i][1]$
    $\mathcal{L}_{min} = \mathcal{L}_{min} + |S_{i-1}| + \mathcal{G}[i][0]$
  **for** $(i = 0 \rightarrow i < |\mathcal{S}|)$ **do**
    **for** $(j = 0 \rightarrow j < |\mathcal{S}[i]|)$ **do**
      $a, b \leftarrow 0$
      **if** $(index > 0)$ **then**
        **if** $(j = 0)$ **then**
          $a = \mathcal{M}ap[index - 1][0] + \mathcal{G}[i][0] + 1$
          $b = \mathcal{M}ap[index - 1][1] + \mathcal{G}[i][1] + 1$
        **else**
          $a = \mathcal{M}ap[index - 1][0] + 1$
          $b = \mathcal{M}ap[index - 1][1] + 1$
      $\mathcal{M}ap[index] \leftarrow (a, b)$
      $index \leftarrow index + 1$
  **return** $\mathcal{M}ap$

Figure 1.12: Construction of the $\mathcal{M}ap$ dictionary.

**requires** The bits mask array $\mathcal{U}(\Sigma)$ of the alphabet $\Sigma$
**procedure** ISEQUIVALENT($d_1, d_2$)
  $s_1, s_2 \leftarrow 0$
  ▷ *perform bitwise* **or** *operation on* $d_1$ *and* $d_2$ *characters*
  **for** $i \leftarrow 1$ **to** $|d_1|$ **do**
    $s_1 \leftarrow s_1 \mid d_1[i]$
  **for** $i \leftarrow 1$ **to** $|d_2|$ **do**
    $s_2 \leftarrow s_2 \mid d_2[i]$
  ▷ *perform bitwise* **and** *operation between* $s_1$ *and* $s_2$ *characters*
  $z \leftarrow \mathcal{U}[s_1]$ & $\mathcal{U}[s_2]$
  **return** $z$

Figure 1.13: Determine whether the degenerate symbols $d_1$ and $d_2$ are equivalent or not.

```
procedure SIMPLISMS(α_i)
    ▷ Build the string 𝒴, the concatenation 𝒮_i, for 1 ≤ i ≤ |𝒮|
    𝒴 ← S_1 S_2 ... S_k
    ▷ Initialize the lists of occurrences, candidate matches and previous matches
    𝒪 ← ∅; 𝒞 ← ∅; ℋ ← ∅
    start, end, ℋ[|ℋ|] ← α_i
    𝒞[|𝒞|] ← ℳ
    for (j = 1 → j < |ℳap|) do
        found ← false
        start ← start + ℳap[j].a − ℳap[j − 1].a
        end ← min(start + ℳap[j].b − ℳap[j].b + 1, |ℱ| − ℒ_max)
        pos ← ∅
        for (q = start → q < end) do
            if (𝒴[j] = ℱ[q]) then
                for all (g ∈ 𝒞[j − 1]) do
                    if (ℳap[j].a − ℳap[j − 1].a ≥ q − g)
                        and (ℳap[j].b − ℳap[j − 1].b ≥ q − g) then
                        ℋ[|ℋ|] ← q
                        found = true
        if found = false then
            break
        else
            𝒞[|𝒞|] ← ℳ
    if |𝒞| = |ℳap| then
        𝒪[|𝒪|] ← 𝒞[0][0])
    return 𝒪
```

Figure 1.14: Determine whether or not the structured motif $\mathcal{M}$ exists in the search context starting at position $\alpha_i$.

```
procedure π-TABLE(S_1)
    π ← 0^{|S_1|}; k ← 0; π[0] ← −1
    for i ← 1 to |S_1| do
        k ← π[i − 1]
        while k ≥ 0 do
            if S_1[k] = S_1[i − 1] then
                break
            else
                k ← π[k]
        π[i] ← k + 1
    return π
```

Figure 1.15: Compute the failure function table $\pi$ for the first seed $S_1$ of the structured motif $\mathcal{M}$.

```
requires the table π
procedure KMPSEARCH (F, S_1)
    α ← ∅; i ← 0; k ← −1
    while i < |F| − L_max do
        if k = −1 then
            i ← i + 1; k ← 0
        else if F[i] = S_1[k] then
            i ← i + 1; k ← k + 1
            if k = |S_1| then
                [|α|] ← i − S_1
                k ← π[k − 1]
        else
            k ← π[k]
    return α
```

Figure 1.16: Compute the list $\alpha$ of starting positions of the first seed $S_1$ of the structured motif $\mathcal{M}$ in the $\ell$-factor $\mathcal{F}$ of the sequence $\mathcal{X}$ (KMP).

```
procedure BUILDBADCHARACTERSHIFT(p)
    badCharacterShift ← 0^|Σ|
    for c = 0 → c < |badCharacterShift| do
        badCharacterShift[c] ← |p|
    for i = 0 → i < |p| − 1 do
        badCharacterShift[p[i]] ← |p| − i − 1
    return badCharacterShift
```

Figure 1.17: Build Bad Character Shift Array of pattern $p$.

```
procedure FINDSUFFIXES(p)
    f ← 0; g ← |p| − 1
    suffixes ← 0^(|p|+1); suffixes[|p| − 1] ← |p|
    for i = |p| − 2 → i ≥ 0 do
        if (i > g) and (suffixes[i + |p| − 1 − f] < i − g) then
            suffixes[i] ← suffixes[i + |p| − 1 − f]
        else
            if i < g then
                g ← i
            f ← i
            while (g ≥ 0) and (p[g] = p[g + |p| − 1 − f]) do
                g ← g − 1
            suffixes[i] ← f − g
    return suffixes
```

Figure 1.18: Find suffixes of pattern $p$.

```
requires The suffixes array suffixes
procedure BUILDGOODSUFFIXSHIFT(p)
    goodSuffixShift ← 0^{|p|+1}
    for i = 0 → i < |p| do
        goodSuffixShift[i] ← |p|
    j ← 0
    for i = |p| − 1 → i ≥ −1 do
        if i = −1 or suffixes[i] = i + 1 then
            for (j = 0 → j < |p| − 1 − i do
                if goodSuffixShift[j] = |p| then
                    goodSuffixShift[j] ← |p| − 1 − i
    for i = 0 → i ≤ |p| − 2 do
        goodSuffixShift[|p| − 1 − suffixes[i]] ← |p| − 1 − i
    return goodSuffixShift
```

Figure 1.19: Build Good Suffix Shift Array of the pattern $p$.

```
procedure BOYERMOORESEARCH(F, S_1)
    α ← ∅
    index ← 0
    while (index ≤ |F| − |S_1|) do
        for unmatched = |S_1| − 1 down to(unmatched ≥ 0) do
            if S_1[unmatched] = F[unmatched + index] then
                if unmatched < 0 then
                    α[|α|] ← index
                    index ← index + mgoodSuffixShift[0]
                else
                    t ← |p| + 1 + unmatched
                    a ← goodSuffixShift[unmatched]
                    b ← badCharacterShift[F[unmatched + index]] − t
                    index ← index + max(a, b)
```

Figure 1.20: Compute the list $\alpha$ of starting positions of the first seed $S_1$ of the structured motif $\mathcal{M}$ in the $\ell$-factor $\mathcal{F}$ of the sequence $\mathcal{X}$ (Boyer-Moore).

# Article: # 2

# On the Repetitive Collection Indexing Problem

In large data sets such as genomes from a single species, large sets of reads, and version control data it is often noted that each entry only differs from another by a very small number of variations. This leads to a large set of data with a great deal of redundancy and repetitiveness.

In this article, we propose an indexing structure for highly repetitive collections of sequence data based on a multilevel $q$-gram model. In particular, the proposed algorithm accommodates variations that may occur in the target sequence with respect to the reference sequence.

---

## 2.1 Introduction

Sequencing the whole Human Genome was a major challenge in biological research and was a celebrated breakthrough when it was completed. The goal was to obtain a consensus sequence accounting for the common parts of the genomes of all humans.

Storing genetic sequences of many individuals of the same species promises new discoveries for the whole field of biology, and the low cost acquisition of an individual human genome gives way to "personalized medicine", making use of one's individual genetic profile to tailor treatment to specific needs.

The human genome consists of around 3 billion base pairs (bps), consisting of 23 chromosomes with lengths ranging from about 33 to 247 million bps [HHXZ10]. If a researcher or physician is dealing with many human genomes, then there is a challenge to store, communicate, and manipulate those genomes. Data structures such as the one introduced in this paper can address the storage and querying challenges. DNA sequences within the same species are highly repetitive, for instance, the mutation rate between two random individuals is limited to 1% on average for humans [AIW11, JW04], and with a large set the global difference will only be around 10% including non-coding segments [HHXZ10].

This poses interesting research challenges to efficiently store and access the data. Due to the highly repetitive nature of the sequences, a delta (difference) representation that encodes the differences between two human genomes can be quite small; although a reference sequence is still required to retrieve the information from delta representations. Most classic data compression techniques are not well prepared to deal with the tremendous redundancy found in genomes of the same species.

Flexible and efficient data analysis on such data sets is possible using suffix trees. However, suffix trees occupy $\mathcal{O}(n \log n)$ bits, which very soon inhibits in-memory analysis. Recent advances in full-text self-indexing reduce the space of suffix tree to $\mathcal{O}(n \log \sigma)$ bits, where $\sigma$ is the alphabet size and $n$ is the cardinality of the sequence (also see [BYG96] and [FM00]).

In practice, the space reduction seen on a Human Genome is more than 10-fold [MNSV09]. However, this reduction factor remains constant when more sequences are added to the collection. This causes problems if you wish to store and query a large number of sequences.

Methods like those in *ppmdi*, *gzip* and *bzip2* will not take advantage of the repetitiveness if the repeats are large as they search for repetitions in a bounded window of the text. Other algorithms, like *p7zip*, use a larger buffer and are very successful, yet are unable to decompress individual sequences [CFMPN10].

Classical compression methods tailored for DNA, such as *GenCompress* , *Biocompress*, *Fact* and *GS Compress* have only moderate success [CFMPN10].

Finding matches with an *LZ77* variant with a sliding window would require a multigigabyte buffer, not counting the match-finding structures. Using a context-based statistical coding (e.g., PPM) may efficiently exploit the repetitions only if the considered context is long enough [GD11].

Our method is based on the observation that two sequences share a certain number of $q$-grams if the edit distance between them is within a certain threshold. Moreover, since there are only four letters in the DNA alphabet, we know that the number of all combinations of $q$-grams in a DNA sequence is $4^q$.

In particular, the proposed algorithm accommodates variations that may occur in the target sequence with respect to the reference sequence.

## 2.2 Our approach

In this study, we describe another solution to the compression of a set of genomic sequence data set which compresses the data set based on comparing it with a reference sequence.

We assume that each target string $\mathcal{S}_i^t$ in the given collection is aligned with the reference sequence $\mathcal{S}^r$, e.g., two sequences $\mathcal{S}^r$ and $\mathcal{S}_i^t$ can be represented as $\mu_1\alpha_1\mu_2 \cdot \cdot \mu_k\alpha_k\mu_{k+1}$ and $\lambda_1\alpha_1\lambda_2 \cdot \cdot \lambda_k\alpha_k\lambda_{k+1}$, respectively, where $\alpha_i$'s are common chunks and $\mu_i$'s and $\lambda_i$'s are chunks different from the other string.

A differentially compressed set is a set where a single reference sequence is stored, along with information about the difference between this sequence and the rest of the set. To evaluate the suggested selection method, the compression of the differences type, locations and the size of the compressed set are examined, as is explained in the next section. Although the general framework of relative differential compression is not new in this context, we add some new ideas to the existing algorithms. Our study defines an efficient data structure for storing genomic sequences and a fast algorithm

to randomly access sub-sequences.

The proposed data structure is built through the following steps, which we will describe in more detail in the following section.

(1) Create the empty variations lookup table and $q$-gram dictionary.

(2) Partition the reference sequence $\mathcal{S}^r$ into fixed length sub-sequences.

(3) For each subsequence we generate the set of $q$-grams.

(4) Update the variations lookup table of the sub-sequences.

(5) For each $q$-gram we generate a computer word (signature).

(6) Update the $q$-grams dictionary.

(7) Repeat the steps (2) - (6) for each target sequence $\mathcal{S}_i^t$ in the collection.

## 2.3 Definitions

**Definition 39 ($q$-gram)** *For a given alphabet $\Sigma$, let $\boldsymbol{s}$ be a string, where $\boldsymbol{s} \in \Sigma^+$ and $q$ is a positive integer number, a $q$-gram of $\boldsymbol{s}$ is a pair $(g, i)$, where $g$ is the factor (of length $q$) of $\boldsymbol{s}$ starting at the $i$-th position, that is $g_i = \boldsymbol{s}[i \,..\, i+q-1]$, the set of $q$-grams of $\boldsymbol{s}$, denoted by $\mathcal{G}(\boldsymbol{s}, q)$, is obtained by sliding a window of length $q$ over the the string $\boldsymbol{s}$. There are total of $|\boldsymbol{s}| - q + 1$ $q$-grams in $\mathcal{G}(\boldsymbol{s}, q)$.*

A filter is an algorithm that quickly discards some parts of the text based on some filter criterium, leaving the remaining part to be checked with a proper (online) approximate string matching algorithm.

The $q$-gram similarity of two strings is the number of $q$-grams shared by the strings, which is based on the following lemma.

**Lemma 2.3.1 (The $q$-gram lemma [Ukk92])** *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be strings with the edit distance $\delta(\boldsymbol{x}, \boldsymbol{y})$. Then, the $q$-gram similarity of $\boldsymbol{x}$ and $\boldsymbol{y}$ is at least $Q_{sim} = max(|\boldsymbol{x}|, |\boldsymbol{y}|) - q + 1 - (q \times \delta(\boldsymbol{x}, \boldsymbol{y}))$. Given strings $\boldsymbol{x}$ and $\boldsymbol{y}$, let an occurrence of $x[1 \,..\, i]$ with at most $k$ differences end at position $j$ in $\boldsymbol{y}$. Then at least $i + 1 - (k + 1) \times q$ of the $q$-grams in $x[1 \,..\, i]$ occur in the substring $\boldsymbol{y}[j - i + 1 \,..\, j]$.*

The value $Q_{sim}$ in the lemma is called the threshold and gives the minimum number of $q$-grams that an approximate match must share with the pattern, which is used as the filter criterium [JU91].

Given two strings $\boldsymbol{s}$ and $\boldsymbol{s'}$, and threshold $h$ such that $\delta(\boldsymbol{s}, \boldsymbol{s'}) \leq h$, similarly as in [UW93]. We denote the list (array) of differences as $\mathcal{H}[0\,..\,h-1]$, where $0 < h < n$ and $h << n$. Array $\mathcal{H}$ stores triplets such that for each triplet $(o, p, c) \in \mathcal{H}[i]$, where $0 \leq i < h$, $\mathcal{H}[i]\,.\,o$ represents the edit operation (0 for replacement, 1 for insertion, $-1$ for deletion) applied in position $\mathcal{H}[i]\,.\,p$ of $\boldsymbol{s}$. In the case of replacement or insertion, $H[i]\,.\,c$ represents the new symbol (base). The array is constructed in such a way that it is already sorted by $\mathcal{H}[i]\,.\,p$, i.e., $\mathcal{H}[i]\,.\,p \leq \mathcal{H}[i+1]\,.\,p$, for all $0 \leq i < h-1$. As an example, see Figure 2.1. Notice that $\mathcal{H}$ describes how sequence $\boldsymbol{s}$ can be transformed to $\boldsymbol{s'}$. The array $\mathcal{H}$ can be computed in $\mathcal{O}(hn)$ time and space [GBL95].

**Problem 2.3.2 (Repetitive Collection Indexing Problem)** *Given a collection $\mathcal{C}$ of $m$ sequences where $\mathcal{S}_i \in \mathcal{C}$ such that $|\mathcal{S}_i| = n'$ for each $1 \leq i \leq m$ and $\sum_{i=1}^{m} |\mathcal{S}_k| = n$, where each sequence $\mathcal{S}_2, \mathcal{S}_3, \ldots, \mathcal{S}_m$ (we call them the target sequences $S_k^t$) contains $h$ mutations from the base sequence $\mathcal{S}_1$ (we call it reference sequence $\mathcal{S}^r$), this means $\delta(\mathcal{S}^r, \mathcal{S}_k^t) \leq h$ for $1 < k \leq m$. The repetitive collection indexing problem is to store $\mathcal{C}$ in as small amount of space as possible such that the following operations are supported as efficiently as possible:*

*Given a pattern $\boldsymbol{p}$ of length $|\boldsymbol{p}| = \ell$ over an alphabet $\Sigma$ and integer threshold $h > 0$, find whether $\boldsymbol{p'}$ occurs in the collection $\mathcal{C}$ ($\mathcal{S}_i$, for $1 \leq i \leq m$), where $\delta(\boldsymbol{p'}, \boldsymbol{p}) \leq h$. The set of positions of $\boldsymbol{p'}$ occurrences in $\mathcal{S}$ are defined as:*

$$Occ(\boldsymbol{p}, \mathcal{S}) = \{1 + |\boldsymbol{u},\ \exists\, \boldsymbol{u},\ \boldsymbol{v},\ \mathcal{S} = \boldsymbol{u}\boldsymbol{p'}\boldsymbol{v}\ ,\ \textit{where}\ \delta(\boldsymbol{p'},\ \boldsymbol{p}) \leq h|\}$$

- $\boldsymbol{exists}(\boldsymbol{p}, \mathcal{S})$ *returns true iff* $Occ(\boldsymbol{p}, \mathcal{S}) \neq \emptyset$.
- $\boldsymbol{count}(\boldsymbol{p}, \mathcal{S})$ *returns* $|Occ(\boldsymbol{p}, \mathcal{S})|$.
- $\boldsymbol{locate}(\boldsymbol{p}, \mathcal{S})$ *returns the ordered set* $Occ(\boldsymbol{p}, \mathcal{S})$.
- $\boldsymbol{extract}(\mathcal{S}, \ell, \ell')$ *extracts the substring* $\mathcal{S}[\ell\,..\,\ell']$.

## 2.4 Algorithm

### 2.4.1 Index construction algorithm

**Step 1.** The algorithm extracts the factors of length $\ell$, called $\ell$-factors, from the set of sequences such that consecutive $\ell$-factors overlap with each other by $q-1$. The overlap is to cover all the $q$-grams $g_i$, where $q$ is the length of the each $q$-gram extracted in the next step.

**Step 2.** The algorithm processes the $\ell$-factors obtained in Step 1. For each $\ell$-factor $\boldsymbol{s}$ occurring $f$ times in a sequence $\mathcal{S}_k^t$ at offsets $\{o_1, \ldots, o_f\}$, a posting $\langle k, \{o_1, \ldots, o_f\}\rangle$ is appended to the posting list of the $\ell$-factor $\boldsymbol{s}$.

**Step 3.** The algorithm extracts $q$-grams from the set of $\ell$-factors obtained in Step 1 by using the sliding window technique, for each $\ell$-factor $\boldsymbol{s}$ of length $\ell$ of $\mathcal{S}^r$ extract the $q$-grams $g_i = \boldsymbol{s}[i\,..\,i+q-1]$.

**Step 4.** The algorithm builds the $q$-grams dictionary, using the $q$-grams obtained in Step 3. For each extracted $q$-gram, $g$, generate the signature $\varsigma(g)$.

**Step 5.** The algorithm fills the $q$-gram dictionary, using the $q$-grams obtained in Step 3. For each $q$-gram $g$ occurring $f$ times in $\ell$-factor $\boldsymbol{s}$ at positions $\{p_1\,..\,p_f\}$ an entry $\langle \boldsymbol{s}, \{p_1\,..\,p_f\}\rangle$ is appended to the entry list of $\varsigma(g)$ in the $q$-gram dictionary.

#### Processing the Reference Sequence

We will use word-level parallelism by packing the $q$-grams into computer words. These words will be referred to as signatures. The signature $\varsigma(\boldsymbol{s})$ of a string $\boldsymbol{s}$ is obtained by transforming the string to its binary equivalent. This is done by using 2-bits-per-base encoding of the DNA alphabet, and storing its decimal value in a computer word.

We extract a set of factors of length $\ell$ of $\mathcal{S}^r$ (the first sequence in the collection), $\mathcal{S}_i^r = \mathcal{S}^r[i\,..\,i+\ell-1]$ such that consecutive $\ell$-factors overlap with each other by $q-1$). For each factor $\boldsymbol{t}$ of length $\ell$ of $\mathcal{S}^r$, we extract the $v$ equal factors of length $j$ of $\boldsymbol{t}_i^j = \boldsymbol{t}[i+j\frac{\ell}{v}\,..\,i+(j+1)\frac{\ell}{v}-1]$, for all $0 \leq i < n-\ell+1$, $0 \leq j < v$. We build an array of linked lists $\mathcal{L}[s]$, for all $0 \leq \boldsymbol{s} < 2^{2\frac{\ell}{v}}$. We compute $\varsigma(\boldsymbol{t}_i^j)$, the signature of $\boldsymbol{t}_i^j$, and insert the pair $(u, b)$ in $\mathcal{L}[\varsigma(\boldsymbol{t}_i^j)]$, where $u$ represents the offset of $\boldsymbol{t}_i^j$ in $\boldsymbol{t}$ and $b$ indicates

whether the factor is mapped to the reference sequence ($v = 1$) or not ($v = 0$). Thus, the pairs $(u, v) \in \mathcal{L}[\varsigma(\boldsymbol{t}_i^j)]$, for all $0 \leq \varsigma(\boldsymbol{t}_i^j) < 2^{2\frac{\ell}{v}}$, are sorted by $u$.

### Processing the Target Sequences in the Collection

Similarly, for the next (target) sequence $\mathcal{S}_k^t$, where $1 < k \leq m$, we extract the set of factors of length $\ell$ of $\mathcal{S}_k^t$, then we compute the array $\mathcal{H}$ and a new array $\texttt{OP}$ , where $\texttt{OP}[i] = \sum_{j=0}^{i-1} \mathcal{H}[j] \centerdot o$ represents the operation sum of $\mathcal{H}[i] \centerdot o$, for all $0 \leq i < h$. An example of this can be seen in Figure 2.1 which shows how to change the sequence $\boldsymbol{t}$ into $\tilde{\boldsymbol{t}}$.

Assume that we have an edit operation $\mathcal{H}[\lambda] \centerdot o$, for some $0 \leq \lambda < h$, in position $\mathcal{H}[\lambda] \centerdot p = p$ of $\boldsymbol{t}$. We compute the signatures of all the $\frac{\ell}{v}$ factors (smaller grams) of $\boldsymbol{t}$, affected by operation $\mathcal{H}[\lambda] \centerdot o$. Let $\varsigma_j$ be the signature of the $j$-th affected factors ($q$-gram), and $\mathcal{L}[s_j][q]$ the $q$-th element of the linked list $\mathcal{L}[s_j]$. For each edit operation $\mathcal{H}[\lambda] \centerdot o$, for all $0 \leq \lambda < h$, the affected factors are defined as follows.

- **Replacement:** $\mathcal{S}^r[p - \frac{\ell}{v} + 1 + i \,..\, p + i]$**, for all** $0 \leq i < \frac{\ell}{v}$**.**
- **Insertion:** $\mathcal{S}^r[p - \frac{\ell}{v} + 1 + i \,..\, p + i]$**, for all** $0 \leq i < \frac{\ell}{v} - 1$**.**
- **Deletion:** $\mathcal{S}^r[p - \frac{\ell}{v} + 1 + i \,..\, p + i]$**, for all** $0 \leq i < \frac{\ell}{v}$**.**

Then we insert the newly created signatures in the variations dictionary, we build a an array of liked lists $\mathcal{M}[\varsigma(\boldsymbol{s})]$, where $\boldsymbol{s} = \boldsymbol{t}_i^j$, to hold the $q$-grams affected by operation $H[\lambda]$.$o$. We compute $\varsigma(\boldsymbol{s})$, the signature of $\boldsymbol{s}$ and insert the pair $(u, b)$ in $\mathcal{M}[\varsigma(\boldsymbol{s})]$, where for each $(u, b) \in \mathcal{M}[\varsigma(\boldsymbol{s})]$, $u$ represents the offset of $\boldsymbol{t}_i^j$ in $\mathcal{S}^r$ and $(b = 0)$ indicates that the factor is not mapped to the reference sequence. Thus, the couples $(u, b) \in \mathcal{M}[\boldsymbol{s}]$, for all $0 \leq s < 2^{2\ell}$, are sorted by $u$.

**Compressing the Index:** We store the relative distance for the posting list (compress each list separately to enable partial decompressing). We use grammar based compressing to compress $q$-grams to enable detecting repetition in the text and allow fast local compression.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | $\mathcal{H}.p$ | $\mathcal{H}.o$ | OP |
|---|---|---|---|---|---|---|---|---|---|---|---|
| string $\boldsymbol{t}$ | $C$ | $A$ | $T$ | $G$ | $G$ | $A$ | $C$ | $A$ | 1 | 0 | 0 |
| | $C$ | $G$ | $T$ | $G$ | $G$ | $A$ | $C$ | $A$ | 2 | -1 | -1 |
| | $C$ | $G$ | $G$ | $G$ | $A$ | $C$ | $A$ | | 7 | 1 | 0 |
| string $\tilde{\boldsymbol{t}}$ | $C$ | $G$ | $G$ | $G$ | $A$ | $C$ | $G$ | $A$ | | | |

Figure 2.1: Operation Sum Table for changing sequence $\boldsymbol{t}$ to $\tilde{\boldsymbol{t}}$

### 2.4.2 Pattern Matching

The filter we use is based on counting the number of $q$-grams common to both the pattern and the current text window. A pattern of length $m$ has $(m-q+1)$ overlapping $q$-grams. Each error can alter at most $q$ of these $q$-grams and therefore $(m-q+1-k\times q)$ pattern $q$-grams must appear in any occurrence for it to be valid [Nav01].

In the index, for each $q$-gram $g$ of the sequence $\mathcal{S}_k^t$ in the collection $\mathcal{C}$, we have an inverted list of the id's of all the $\ell$-factors that contain this gram. If a gram appears in a $\ell$-factor multiple times (with different positions), the factor id will appear multiple times on the inverted list of this gram, with the different positions.

Let $\boldsymbol{s}$ and $\boldsymbol{p}$ be strings of length $n$ and $m$ respectively, such that $\delta(\boldsymbol{p},\boldsymbol{s}) = k$. Then any substring of $\boldsymbol{s}$ (including the string $\boldsymbol{s}$) have at least $m - q + 1 - \delta \times q$ common $q$-grams with $\boldsymbol{p}$. Finding the set of $q$-grams can be done in $\mathcal{O}(|G(\boldsymbol{p},q)|\log v)$, where $v$ is the number of unique signatures in the index and $G(\boldsymbol{p},q)$ is the set of $q$-grams of $\boldsymbol{p}$ (From Definition 39, $|G(\boldsymbol{p},q)| = |\boldsymbol{p}| - q + 1$). Then using the $q$-gram lemma (Lemma 2.3.1) we can tell if there is a match with edit distance $\delta \leq k$ by simply checking the number of common $q$-grams which occur in each sequence in the collection.

## 2.5 Complexity analysis

The proposed algorithm consists of two main steps:

1. Compute the edit transcript between the reference sequence and each other sequence in the collection.

2. Creating a dictionary of the computed differences.

The $q$-gram index is constructed in $\mathcal{O}(n)$. The array $\mathcal{H}$ of the edit operations can be computed in $\mathcal{O}(nh)$ [FHIP10].

The array OP of the operations sums can be computed in $\mathcal{O}(h)$ time from $\mathcal{H}$.

By using a technique similar to Karp-Rabin [KR87] when generating the signatures for every $q$-gram the total expected time is kept as $\mathcal{O}(n)$ instead of $\mathcal{O}(nq)$ as the signature of the next $q$-gram can be obtained in $\mathcal{O}(1)$. The next signature can be computed by performing a left shift on the previous signature and appending the extra character to the end of the signature, taking $\mathcal{O}(1)$.

The main components in the proposed structure are the vocabulary list and occurrences lists. The vocabulary list stores the list of distinct $q$-grams that appear in the collection of sequences, while the occurrences list contains, for each $q-gram$ found in the vocabulary, the list of the positions where that $q-gram$ appears.

First, we will consider the size the vocabulary list, to determine the number of different $q$-grams in an arbitrary text, consider that there are $|\Sigma|^q$ different possible $q$-grams and $n$ $q$-grams in the given text (of length $n$). The probability of a $q$-gram to be found is $1/|\Sigma|^q$. Therefore, the probability of a $q$-gram not being selected in $n$ attempts is $((1-1/|\Sigma|^q)^n)$. Hence, the average number of $q$-grams selected in the $n$ attempts is $|\Sigma|^q(1-(1-1/|\Sigma|^q)^n) = \theta(|\Sigma|^q(1-e^{-n/|\Sigma^q|})) = \theta(min(n,\sigma^q))$.

Second, we consider the lists of occurrences. Since we index all positions of all $q$-grams, the space requirements are $O(n)$. If block addressing is implemented (blocks of size $\ell$, the number of blocks is $b = n/\ell$), we consider that there is an entry in the list of occurrences per different $q$-gram mentioned in each different block. So, each block has $(min(\ell, |\Sigma^q|))$ different $q$-grams. To get the total for all blocks, we multiply this by the number of blocks $b = n/\ell$, we have the total size of the occurrence lists is $O(n\ min(1, |\Sigma^q|/\ell))$, The occurrences are found in ascending order, hence each insertion takes $\mathcal{O}(1)$ time. Therefore, the $q$-gram index is built in $\mathcal{O}(n)$. Similarly the variation index can be built in $\mathcal{O}(h)$.

The reduction in space requirements, that is obtained from utilizing block addressing, comes at the expense of extra search costs, to retrieve the exact pattern positions in the collection of sequences a sequential search over the qualifying blocks becomes necessary. The structure is therefore used as a filter to avoid a sequential search over the non-qualify blocks, while the others (qualifying blocks) need to be searched.

# Chapter IV

# Challenges in Arabic computational linguistics

# Article: # 1

# Arabic Morphology Analysis and Generation

This article describes the construction of a lexicon and a morphological description for standard Arabic language. We present a large-scale system that performs morphological analysis and generation of Arabic words. The result, Arabic Morphological Analyzer **AMA**, is a Finite State Transducer, it is based on direct implementation of a comprehensive list of Arabic roots, a dictionary of Arabic morphological patterns and a set of Phonological/Orthographical alternations rules. The output of the system is a large-scale lexicon of inflected/derived forms. Also the system accepts Modern Standard Arabic words and returns morphological analyses and glosses.

---

## 1.1 History

From rock walls and clay tablets through paper to e-book tablets, writing is a system of linguistic symbols which permits one to transmit and conserve information. Writing appears to have developed between the 7-th millennium BC and the 4-th millennium BC, Clay tablets were used in Mesopotamia in the third millennium BC.

Arabic language (العربية) is a central Semitic language, most closely related to Aramaic, Hebrew, Ugaritic and Phoenician, However, not all Semitic languages have equally preserved the features of their common ancestor language. In this respect, Arabic is unique; it has preserved a large majority of the original Proto-Semitic features. In fact, many linguists consider Arabic the most Semitic of any modern Semitic languages in terms of how completely they preserve features of Proto-Semitic, Arabic has lent many words to other languages and borrowed words from many languages.

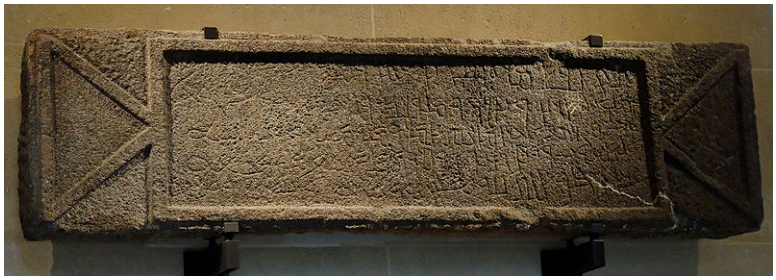


Figure 1.1: Arabic epitaph of "Imru-l-Qays, son of 'Amr, king of all the Arabs", inscribed in Nabataean script. Basalt, dated in 7 Kislul, 223, viz. December, 7 328 AD. Found at Nemara in the Hauran (Southern Syria).

Uncovering the statistics and dynamics of human language helps in characterizing the universality, specificity and evolution of cultures. These days new texts are for the most part created directly in digital form. Also, the explosive growth of the Internet content have resulted in a massive amount of data that is available for research. Apart from the informational content of the texts collected, whether they are created recently or centuries ago, the texts themselves also provide great material for the investigation of human language and its structures.

The influence of Arabic language has been most important around the globe. Arabic is an important source of vocabulary for many languages. Arabic language belongs to the Semitic language family originated in the Arabian Peninsula in pre-Islamic

times, and spread rapidly across the Middle East, it is one of the official languages of the United Nations, the sixth most used language in the world, spoken by over 360 million people (in 2014) and the official language for over 29 countries, in addition to which there are native Arabic speakers scattered all over the world. However, The efforts to improve Arabic information search and retrieval compared to other languages are limited and modest, The barrier to text processing advancements in Arabic is its complicated syntactic and morphological properties which makes it a difficult language to master and explain the lack in the processing tools for that language. Among these properties the complex structure of the Arabic word, the agglutinative nature, lack of vocalization, the segmentation of the text, the linguistic richness, etc...

Because there is not a modern linguistic model for Arabic grammar within the frame of computational linguistics, the results achieved are not comparable with that achieved on other languages.

The Semitic languages are notable for their non-concatenative morphology. Arabic language is a highly inflected language, it has much richer morphology. This raises the need to study the key statistics of Arabic language and the statistical differences between Arabic and other languages on a large scale.

Unlike most languages, Arabic has virtually no means of deriving words by adding prefixes or suffixes to words. Instead, they are formed according to a finite (but fairly large) number of templates applied to roots.

## 1.2 Motivation

In this article, we are primarily concerned with Arabic language. The motivation for working on Arabic is as follows. In formal language theory, the symbols for representing words are inseparable parts of the definition of the language. In human languages, the concept is a little different: an assertion, for example, can have multiple representations, depending on the means of communication and the conventions for recording it.

In the context of linguistics, morphology is the study of word forms. Arabic morphology التصريف is well-known for its richness and complex nature. It has a multi-tiered structure and applies *non-concatenative morphotactics*. In fact, the Arabic mor-

phological complexity is known to have negatively affected the performance of spell checking systems! Words in Arabic are originally formed through the amalgamation of roots and patterns.

To illustrate the complexity of Arabic morphology, inflectional/derivational systems for Arabic words are shown in Figure 1.2. Figure 1.3 shows possible affixes, classifications and morphological templates, all affixes and clitics are optional, and they can be connected together in a series of possible scenarios.

A root in Arabic is a sequence of three or four letters and the pattern is a template of vowels (and non-root letters) with slots into which the radicals of the root are inserted. This process of insertion is called interdigitation [BK03]. The resulting lemmas are then passed through a series of affixations (to express *morpho-syntactic* features) and clitic attachments (as conjunctions and prepositions, for example, are mostly joined to adjacent words in writing) until they finally appear as surface forms. Due to the richness and complexity of Arabic language, there is no corpus, no matter how large, that contains all possible word forms. Given a word in Arabic, one can change its form by adding or removing yet another prefix, suffix, proclitic or enclitic. This is why a morphological generator/analyser is essential in creating an adequate list of possible words in Arabic.

## 1.3 Aspects of Arabic language

Arabic is written and read horizontally from right to left. There are 18 distinct letter shapes (There are no capital letters), which vary slightly depending on whether they are connected to another letter before or after them. The full alphabet of 29 letters is created by placing various combinations of dots above or below some of these shapes. The letters are divided into two groups, called the sun letters (or solar letters) and moon letters (or lunar letters), based on whether or not they assimilate the letter (*Laam* ل) of the preceding definitive article. There are three long vowels included in the 29 letters and six short vowels indicated by marks (Diacritics, or Harakat, الحركات) above and below other letters. The Arabic alphabet letters have up to 4 forms (Isolated, Initial, Medial or Final), the form that a letter takes depends on its position in a given word, and the difference most of the time is very small, like a longer tail to allow it connect with another letter following it, as shown in Figure 1.2.

| ا | ب | ت | ث | ج | ح | خ | د | ذ | ر |
|---|---|---|---|---|---|---|---|---|---|
| Alif | Baa | Taa | Thaa | Jeem | Haa | Kha | Daal | Thaal | Raa |
| ز | س | ش | ص | ض | ط | ظ | ع | غ | ف |
| Zaay | Seen | Sheen | Saad | Dhaad | Taa | Dhaa | Ayn | Ghayn | Faa |
| ق | ك | ل | م | ن | ه | و | ي | ء | |
| Qaaf | Kaaf | Laam | Meem | Noon | Haa | Waaw | Yaa | Hamza | |

Arabic alphabet

| | |
|---|---|
| الشدة | ـّـ |
| السكون | ـْـ |
| الكسرة | ـِـ |
| الضمة | ـُـ |
| الفتحة | ـَـ |
| تنوين كسر | ـٍـ |
| تنوين ضم | ـٌـ |
| تنوين فتح | ـً |

List of Harakat

| |
|---|
| ء |
| ئـ |
| أ = ءَ |
| إ = ءَ |
| ؤ = ءُ |
| ئ = ءِ |

List of Hamza

| Verb/Noun prefix combination | | |
|---|---|---|
| Prefix | Meaning | V/N |
| ب | “in”, “at” | N |
| ف | “therefore” | N & V |
| س | “will” | V |
| ك | “like” | N |
| لـ | “for”, “to” | N & V |
| و | “and” | N & V |
| فب | “therefore in” | N |
| فس | “then will” | V |
| فك | “like” | N |
| فل | “therefore to” | N & V |
| وب | and in, | N |
| وس | “and will” | V |
| وك | “and like” | N |
| ول | “and for” | N & V |

Verb/noun prefix

| Sound verb | سالم |
|---|---|
| Hamzated | مهموز |
| Doubled | مضعّف |
| Weak | معتل |
| Assimilated | مثال |
| Hollow | أجوف |
| Defective | ناقص |

Verb kinds

| The definitive article + prefix | | |
|---|---|---|
| Prefix | Combination | Meaning |
| ال | ال | “the” |
| بال | ب + ال | “in the” |
| فال | ف + ال | “then the” |
| كال | ك + ال | “like the” |
| وال | و + ال | “and the” |
| فلل | ف + ل + ل | “for the” |
| ولل | و + ل + ل | “and for the” |
| فبال | ف + ب +ال | “and in the” |
| فكال | ف + ك + ال | “and like the” |
| وكال | و + ك + ال | “and like the” |

The definitive article prefix

| ألف وصل | أأ = أ = آ |
|---|---|
| ألف خنجرية | ـٰـ = ا |
| ألف مقصورة | ى |

List of Alif

| وزن الفعل الرباعي | |
|---|---|
| المجرد | فَعْلَلَ–يُفَعْلِلُ–فعلل |
| المزيد | تفَعْلَلَ–يَتَفَعْلَلُ–تَفَعْلَلْ |
| | افْعَنْلَلَ–يَفْعَنْلِلُ–افْعَنْلِلْ |
| | افْعَلَلَّ–يَفْعَلِلُّ–افْعَلِلَّ |

Quadrilateral verb forms

| وزن الفعل الثلاثي المزيد | | |
|---|---|---|
| المزيد بحرف | أَفْعَلَ–يُفْعِلُ–أَفْعِلْ | |
| | فَعَّلَ–يُفَعِّلُ–فَعِّلْ | |
| | فَاعَلَ–يُفَاعِلُ–فَاعِلْ | |
| المزيد بحرفين | انْفَعَلَ–يَنْفَعِلُ–انْفَعِلْ | |
| | افْتَعَلَ–يَفْتَعِلُ–افْتَعِلْ | |
| | افْعَلَّ–يَفْعَلُّ–افْعَلَّ | |
| | تَفَاعَلَ–يَتَفَاعَلُ–تَفَاعَلْ | |
| | تَفَعْلَلَ–يَتَفَعْلَلُ–تَفَعْلَلْ | |
| المزيد بثلاثة أحرف | اسْتَفْعَلَ–يَسْتَفْعِلُ–اسْتَفْعِلْ | |
| | افْعَوْعَلَ–يَفْعَوْعِلُ–افْعَوْعِلْ | |
| | افْعَوَّلَ–يَفْعَوِّلُ–افْعَوِّلْ | |
| | افْعَالَّ–يَفْعَالُّ–افْعَالَّ | |

Trilateral unaugmented verb

| وزن الفعل الثلاثي المجرد |
|---|
| فَعَل يَفعَل افْعَل |
| فَعَل يَفعِل افْعَل |
| فَعَل يَفعِل افْعَل |
| فَعِل يَفعَل افْعَل |
| فَعِل يَفعَل افْعَل |
| فَعِل يَفعِل افْعَل |

Trilateral augmented verb

| | الضمير |
|---|---|
| أنا | أ |
| أنا | أ |
| نحن | ن |
| نحن | ن |
| نحن | ن |
| نحن | ن |
| أنتَ | ت |
| أنتِ | ت |
| أنتما | ت |
| أنتما | ت |
| أنتم | ت |
| أنتنّ | ت |
| هو | ي |
| هي | ت |
| هما (مذكر) | ي |
| هما (مؤنث) | ت |
| هم | ي |
| هنّ | ي |

Pronouns list

Figure 1.2: Arabic language structure.

In Arabic, sentences are composed of a number of types (اقسام الكلام) of words (or part of speech). A word in Arabic can be a noun, a verb, or a particle.

- Noun ( اسم), includes nouns, pronouns, adjectives and adverbs, etc..;
- Verb (فعل), all different kinds of verbs and their conjugations including both unaugmented and augmented forms;
- Particle (حرف), particles, articles and conjunctions, etc..;

Words in Arabic (whether verbs or nouns), so as in Semitic languages, are generally based on a *root* (جذر) which uses a sequence of consonants (or radicals, also sometimes referred to as the base letters) to define the underlying meaning of the word.

In Arabic, roots convey a basic meaning which then allow for more complex semantic concepts to be constructed, whether these are verbs or nouns.

Words are formed out of roots not so much by adding prefixes or suffixes, but rather by filling in the vowel and non-root letters between the root radicals to create the required inflection of meaning. They are formed according to a finite (but fairly large) number of morpho-phonological templates/rules.

Roots are classified based on the number of radicals it contains, and often referred to as *Triliteral* (الثلاثية) for 3 letters root and *Quadriliteral* (الرباعية) for 4 letters roots. *Triliteral* roots form the overwhelming majority $7,889$ ($5,184$ unique combinations), and to a lesser extent, *Quadriliteral* approximately $2,000$.

The Arabic letters *Faa*, *Ayn* and *Laam* are typically used as placeholders (ف ع ل) for *Triliteral* and ((ف ع ل ل) for *Quarilatrial* roots) in morphological patterns to denote three different base (radical) letters, the word (فعل) is a prototypical verb that means "to do" or "to act".

Roots where both the second and third radicals are identical are called *doubled*. Roots containing one or more of the radicals *Hamza* (ء), *Waw* (و) or *Yaa* (ي) are called *weak* roots (الفعل المعتل) (in contrast to the *sound* roots (الفعل الصحيح)), verbs that derived from such roots are also called *weak* verbs, the paradigm of such roots must be given special attention, often require special phonological/orthographical rules because these radicals can be influenced by their surroundings/positions.

Some roots fall into more than one of the following main categories, resulting in 30 sub-types in total according to the position and number of the weak radicals in the root.

- Sound roots (الفعل الصحيح السالم).
- Doubled roots (الفعل المضعف).
- Hamzated roots (الفعل المهموز).
- Assimilated roots (الفعل المثال).
- Hollow roots (الفعل الأجوف).
- Defective roots (الفعل الناقص).

Arabic language has six categories (الأوزان) based on the vowel symbol (Diacritic mark) (حركة عين الفعل) on the second radical letter (عين الفعل). These categories (الأوزان) play a crucial role in classifying each word in its appropriate context and usually memorized by the following verse (بيت شعر).

فتح ضم فتح كسر فتحتان كسر فتح ضم ضم كسرتان

*Verbs* that derived from roots consisting of three or four radicals (*Triliteral* or *Quadriliteral* roots) are called *Unaugmented* verb forms. Furthermore, Arabic morphology includes augmentations of the roots, *augmentation* is the procedures for creating *Augmented Forms*, the procedures for creating new verb Forms by adding (1, 2, or 3) letters and (1 or 2) letters to the *Triliteral* and *Quadriliteral*verb forms respectively. There are 12 (identified as I - XII Forms) augmented forms for the *Triliteral* roots and 3 (identified by QI, QII and QIII Forms) augmented forms for the *Quadriliteral* roots, these *augmented* forms are not just inflections of *unaugmented* form of the verb – they are independent verbs in their own right.

It has to be said that not all augmented Forms exist for all roots, and the suggestion that each Form has its own meaning (e.g., that Form II is always causative) is arguable. However, there is agreement that there is a standard procedure for creating

each Form, and that each Form is a new word of different or the same meaning from the *unaugmented* form of the verb.

*Verbs* (الافعال): in Literary Arabic are marked for person (first, second and third), gender(masculine and feminine), number (singular, dual and plural), tense (past, present and imperative) and six moods (indicative, imperative, subjunctive, jussive, shorter energetic and longer energetic).

Verbs cab be Transitive متعدي, so that they require an object (or two) to convey complete meaning or Intransitive لازم, means that the verb does not need an object to make sense. However it is possible to a verb which is both (Transitive and Intransitive) depending on the context.

Verb conjugation is the study of how verbs are derived from a set of base letters and how they conjugate in the different tenses to reflect different grammatical categories, such as gender, plurality, voice, and other aspects. The set of grammatical categories, and the procedures for and the procedures for expressing them, are the same for all forms.

*Nouns* (الاسماء), the terms actually mean the broader part of speech than it means in English. Similar to verbs, nouns can be assigned into categories, and inflicted, based on many considerations such as gender, plurality, mode, state and more. Inflected nouns can be attached to the definite article and/or list of conjunctions, prepositions, particles and pronouns prefixes.

Furthermore, nouns can be divided based on derivational categories.

– **Derived noun (مشتق):** is a word derived from a verb

– **Verbal nouns (or gerund ) (مصدر):** is a word that indicates the occurrence of an action and is free of tense

– **Static nouns (جامد):** is neither a derived noun nor a gerund, these are not derived from anything and nothing is derived from them.

There are 4 types gerunds and 7 types of derived nouns, each one of these types (derived noun and gerunds) comes with 2 sets of (standard and non-standard) patterns and comes with sets of morphological rules that tell us how to construct them.

*Particles* (الحروف)) are the third type (parts-of-speech) in Arabic, include prepositions, conjunctions, interjections, question particles and answer particles, they dont

fall into the morphological templatic system (i.e., they have no patterns) nor do they undergo any *morpho-phonemic* changes. There are relatively few of them and they must be maintained in special dictionary. For more details about Arabic grammars see [AJ87] and [AB07].

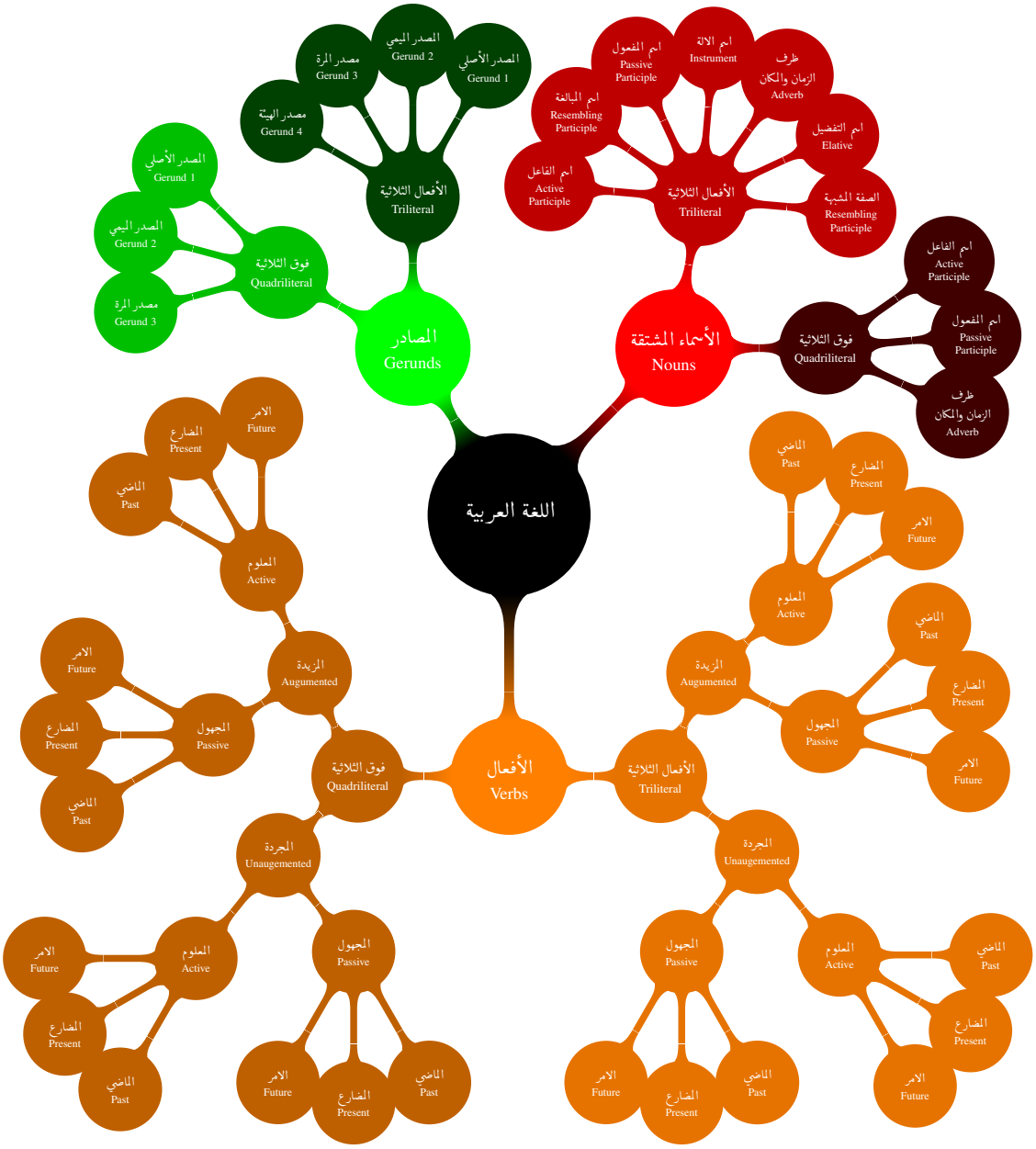



Figure 1.3: Arabic word formation (derived/inflected) tree

## 1.4 Arabic Morphological analyzer (AMA)

In language processing, an Finite State Automaton (FSA) can be used to recognize or generate a specific language defined by all possible combinations of characters (conditional labels) on each of the edges generated by traversing the FSA from the initial state to the end state. Each path from the initial state to a final state can be seen as a mapping between a surface form and its lexical form.

Using an FSA to recognize a morphological realization of a word is useful. However, we also want to return an analysis of that word. To be able to do this, we need a Finite State Transducer (FST).

A finite state transducer (FST) is a special type of finite state automaton that works on two (or more) tapes. Rather than just traversing (and accepting or rejecting) an input string, a transducer works like a sort of "translating machine". It reads from one of the tapes and writes onto the other. In the *translation mode*, an FST translates the contents of its input string to its output string. In the *generation mode*, it accepts a string on its input tape and generates another string on its output tape.

In our proposed Arabic Morophological Analyzer **AMA**, the creation of the surface forms from the roots is best thought of as a three-stage process:

**First stage:** The root and the morphological template are merged together, this is done mainly by replacing the placeholders slots in the template with the root letters;

**Secound stage:** Then prefixes and suffixes for each form are attached to reflect person, gender, number, voice, tense, aspect, mood and state;

**Third stage:** Finally, phonotactic constraints and orthographic normalization rules are applied to produce the final surface forms.

First and second stages are carried out in a systematic way, these two stage are almost identical for all roots. Also the set of grammatical categories, and the procedures for embodying them, are the same for all forms. In the third stage, some additional complications arise with so-called weak roots, at this point, deep phonological/orthographical alterations are carried out, such as gemination, vowelization and substitution.

Now we briefly review the main components of our proposed Arabic morphological analyzer AMA.

► **Template dictionary.** This module stores a set of morphological templates formed according to the Arabic language grammars (see [AJ87] and [AB07] for more details). The pattern morpheme is an abstract template to represent a word as a string of letters including special symbols to mark where root radicals and vocalization are inserted and occupy specified places.

A pattern can include additional consonants and vowels letters usually represented by $C_n$ where $1 \leq n \leq \ell$ where $\ell$ is the length of the pattern.

Note that the morphological template dictionary, contains the templates that surface forms of the words are generated from (and not the words itself as seen in text). It only contains the morphotactic (rules governing the combination of morphemes) and orthographic (spelling) rules. The template dictionary stores all the forms of word structures represented by a template in which roots are accompanied by a sequence of slots in fixed positions, filled by mutually exclusive systems of contrasting affixes.

For example, the template [$\forall + \forall + $ ت $+ \forall + $ ا] represents the X-Form of *Triliteral*-verb augmented by two letters, in the past tense. Similarly, the template [$\forall + \forall$ ت $+ \forall +$ ي] represents the X-Form of *Triliteral*-verb augmented by two letters in present tense. The symbol $\forall$ in the above templates represents the set of letters associated with each template. Thus, by the interdigitation of a root and a pattern surface forms are created.

► **Rules Engine.** The Rules Engine was designed to provide automation based on external rules. The rules were to govern behaviour of verb/noun in response to gemination, vowelization and substitution. De-coupling the rules of the application helps to maintain the list of rules independently. In Arabic when a word is formed (derived or inflicted) from the so-called weak root, some letters may be dropped, changed, doubled or replaced by other letters.

The rules engine whose responsibility is enforcing the rules applies the changes to variables at the generation stage and reverses those changes at the matching stage. An example of such changes is as follows. The conjugation of verbs containing the letter *Waw* (واو) will often require replacing the letter *Waw* (واو) by the letter *Alef* (الف) in the inflicted/derived form.

The main causes of these irregularities are phonological constraints on the letters *Alef*

(الف), *Waw* (واو) and *Yaa* (ياء). Words derived from roots- containing at least one or more of these letters typically present phonological alterations to make the word pronunciation easier. Another cause of these presence of two identical is the presence of two identical adjacent letters in the root(the second and third letters), resulting in what so called doubled roots. Also this phenomena, in some cases, is related to the so-called hamzated roots (roots that contain *Hamza* (الهمزة)), forms that derived from these verbs exhibit different orthographical shapes depending on the surrounding context.

Despite these irregularities, the set of surface forms can be related to a single underlying semantic category.

At this stage, all derived/inflected forms are going through a series of rewrite-rules in the form of regular expressions. The set rules are organized so that the form which requires various phonological alterations are all applied in the correct order.

In summary, AMA is basically a finite state transducer (FST), was obtained by directly implementing a comprehensive list of Arabic roots, a dictionary of Arabic morphological patterns and a set of Phonological/Orthographical alternations rules. It contains a list of 7,889 (5,184 unique radicals combinations) *Triliteral* roots. The total number of generated words (surface-forms) is 260,922,024 as described below:

- 58,463,856 verb, split in to 15,336,216 generated from *unaugmented* forms and 43,127,640 generated from *augmented* forms;
- 73,621,152 *gerunds*;
- 128,837,016 *derived nouns*.

The preformace of morphological analyzers is evaluated on the generated analysis, where an analysis is considered complete all of its identification and notations is fully correct. Note that the analyses, conserns word types, captures the degree of under/over generation of analysis generation. The learnt rules from training (morphologically annotated) data set is then applied to unknown word types in the un-annotated corpus.

The evaluation process quatitivly measures the preformance of morphological analyzers in terms of the most common metrics such as precission, recall, accuracy and F-score.

Another typical approach is to perform application oriented evaluation to predict how the morphological analyzer acts in the context of various kinds of applications, the benchmark presneted in the next article to evalueate the performance of “AMA” as stemmer is an example of such approach.

In both case, the main purpose of the evaluation process is to compare results returned by morphological analyzers implementations using known annotated corpora. This operation gives an idea about the relevance of the results according to the selected corpus.

# Article: # 2

# Novel Arabic Language Stemmer

Arabic information retrieval can be enhanced when the roots (or stems) are used in indexing and searching. Arabic language exhibits a very rich and complicated morphological structure, its unique structure presents many difficult challenges in the stemming process.

Existing Arabic stemmers suffer from high stemming error-rates because it is hard to differentiate between base and affix letters in a word, most of the existing stemmers blindly stem all types of word (verbs and nouns) in the same fashion by stripping off prefixes, infixes and suffixes and using pattern matching with the aid of look up dictionary for identifying the roots.

We present a new stemming technique by augmenting the Arabic Morphology Analyzer **AMA** [AI12a] with new stemming model, derived from morphological and phonological representation of the language in order to minimise both stemming errors and stemming cost.

The proposed stemmer determines the root of a given word, which represents the semantic core of this word, based on comprehensive study of Arabic morphophonology with its basics and intricacies. The proposed stemmer efficiency and effectiveness are evaluated by comparison with a superior root-based *Khoja stemmer* and stem-based *Light*-10 stemmers.

---

## 2.1 Introduction

In linguistic morphology and information retrieval, stemming is the process for reducing inflected (or derived) words to their stem, base or root form (generally a written word form). The stem need not be identical to the morphological root of the word; it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. Algorithms for stemming have been studied in computer science since 1968 [FF03].

Stemming is defined as the process of conflating of all variations (surface forms) of specific words to a single form called the root or stem for example, “stemmer”, “stemming”, “stemmed” will be reduced to “stem”.

Many search engines treat words with the same stem as synonyms as a kind of query broadening, a process called conflation [SAB11]. Stemming programs are commonly referred to as stemming algorithms or stemmers.

Stemming algorithms are an intrinsic component in document retrieval systems, they have been applied and tested in building of information retrieval systems, for many languages among which for English is the well known *Porters stemmer*.

The process of selecting the representation or index terms constitutes a major operation and technique applied in information retrieval systems.

The main advantages of applying word stemming in the indexing process is that it helps in reducing the size of the index terms, improving the morphological search, and also help in improving the degree of relevancy in retrieving documents.

Practical stemming algorithms for the Arabic language are not widely available. The existing algorithms are either generic in nature, derived from algorithm originally designed for other languages (such as “Suffixes-Striping” or “Soundex”) or lack in the morphological aspect of getting to the correct root/stem or word in Arabic laguage.

## 2.2 Prior work

Root indexing and light stemming are the most widely used indexing mechanisms in Arabic Information Retrieval (See [AF02] for in details comparison). Extensive research was conducted to investigate the effect of these two approaches (Root based and Light stemming), adopted as an indexing mechanism, on improving Arabic Informa-

tion Retrieval . As a result, several versions of Arabic light stemmers with various combinations of suffixes and prefixes were discussed in the literature. Many researchers debated that roots were the best way of indexing Arabic documents.

A well-known root-based stemmer is the Khoja stemmer, presented by Khoja [Kho01]. The Khoja algorithm removes suffixes, infixes and prefixes and uses pattern matching with the aid of a dictionary to extract the roots.

Although the algorithm suffered from problems especially with proper nouns, broken plurals (i.e., nouns that do not follow any rule for pluralizing), and verbs, the Khoja's algorithm showed superiority over previous work in root detection algorithms [AF02].

The Khoja stemmer needs constant maintenance to track newly discovered words, thus many attempts were made to enhance the Khoja stemmer. Taghva et.al [TEC05] presented an improvement to the Khoja stemmer by eliminating the need of a dictionary; the new stemmer performed equivalently to the Khoja stemmer in the same environment. However, scholars did not adopt the ISRI system due to its complicated nature.

The Khoja stemmer (root-based) tends to stem morphologically related words (but not necessarily semantically related words) and as a result, has a high over-stemming error rate. Additionally, when dealing with nouns, the Khoja stemmer fails to conflate the words in the same conceptual group causing a high under-stemming error.

Another category of Arabic stemming algorithms are the stem-based (Light) algorithms [LBC02] created a group of light stemmers including Light-$1, 2, 3, 8$ and Light-$10$. The latest, Light-$10$ stemmer, is shown to outperform the previous versions of light stemmer. Light stemming does not deal with patterns or infixes, it simply strips off prefixes and/or suffixes. However, the brutal stripping off of a fixed set of prefixes and suffixes causes many stemming errors, especially where it is not easy to differentiate between the root letters and the attached letters. Therefore, the Light$10$ stemmer, achieved a very low recall average because it could not retrieve all the documents in many cases because of the unsupervised removal of a fixed set of prefixes and suffixes.

Tengku Mohd T. et.al, [SAB11] presented a rule-based Arabic stemming algorithm. This algorithm will try to find all the valid possible roots for a given word. The algorithm will check for the root validity by using the hashing technique to search for it in the root dictionary.

Darwish et.al, [DO07] presented a modified light stemmer "Al-stem" with an extended prefixes and suffixes lists. In a monolingual IR environment, the Light-10 had a higher average precision in comparison to the "Al-stem".

Al-Shammari et.al, [ASL08] presented novel algorithm automatically creates it is own list of proper nouns, and compound words based on the processed corpus to reduce both stemming error and stemming cost.

## 2.3 Definitions

Recall, the Hamming distance (Definition 5) between two strings of equal length is the number of positions with mismatching characters.

**Definition 40 (Masked Hamming Distance)** *Given two strings, $u$ and $v$, of equal length $n$ where $u, v \in \Sigma^n$, we denote by "$*$" a don't care symbol, "$*$" has the property of matching any single character $\alpha$ such that $\alpha \in \Sigma$.*

*Assume that $u$ and $v$ contain some occurances of "$*$". We define the* Masked Hamming Distance *between $u$ and $v$ as follows. We say that two symbols $u[i]$, $v[i]$, for $i \in [1..n]$, match if and only if*

- $u[i] = v[i]$ *or*
- *either $u[i]$ or $v[i]$ is a don't care character.*

In other words, the *Masked Hamming distance* between two strings $u$ and $v$ is the number of mismatching characters after excluding all the positions $i$, for $i \in [1..n]$, where $u[i] = *$ or $v[i] = *$. In this way, the pattern approximately matches the text at given locations.

## 2.4 Our approach

Words constructed from the same root constitute what is traditionally called a morpho-semantic field, where semantic attributes are assigned through patterns governed by morphological rules. The meaning that is inherent in the root is shared by all words

in this field. However, the patterns that produce these words make them semantically distinguished.

In Arabic, the word is split into four morphological segments: a prefixing conjunction, the main stem (a verb or noun form), and two suffixes (an attached subject pronoun and an attached object pronoun). In morphology, one usually has to model two principally different processes:

- Morphotactics (how to combine word forms from morphemes).
- Phonological/Orthographical alternations rules (when a word is formed (derived or inflicted) from a root, some letters may be added, dropped or replaced by other letters).

Typically, stemmers work by either relying on a lookup table that consists of surface forms and root form relations or by applying suffix/prefix stripping approaches guided by some grammatical rules. Instead, in our proposed stemmer, we take a different approach, the proposed stemmer works by identifying the morphological template that the input word form, to be stemmed, belong to, once the template (of the given input word) is identified, it becomes easier to identify the base letter from the attached letters in the word. In another words, we reduce the problem of stemming a given word (surface form) to, simply, finding the morphological template of the given word.

Here we list several determined (though very useful) rules for identifying the morphological template of a given word from Arabic language grammars.

- the root and the template length.
- the set of letters that can form the root.
- the set of letters that can be attached (الحروف المزيدة), not every letter in the alphabet may act as extra (can be attached). In fact, the letters that can are memorized the following words: (سألتمونيها)
- letters proximity/distance.
- combination of prefix/suffix (with verbs or nouns).
- the number and locations of don't care letters.

Furthermore, Arabic language exhibits a restricted morphological system on how words are structured, based on their type (noun or verb). Generally, words are constructed according to one of the following two templates:

**Verb forms structure**

- prefix (verb) سوابق الافعال $and/or$
- the present tense attachment letters حروف المضارعة and
- verb form صيغة الفعل and/or
- enclitic pronouns (subject) suffix ضمائر الرفع and/or
- enclitic pronouns (first/second object) suffix ضمائر النصب and/or
- person/gender/number suffix ملحقات المتكلم والجنس والعدد.

**Noun forms structure**

- prefix (noun) سوابق الاسماء and/or
- the definitive the ال التعريف and
- derived or verbal noun form صيغة الاسم المشتق او المصدرand/or
- enclitic pronouns suffix ضمائر الرفع and/or
- enclitic pronouns suffix ضمائر النصب و الجر and/or
- enclitic pronouns (proposition) suffix ضمائر الاضافة and/or
- person/gender/number suffix ملحقات المتكلم والجنس والعدد.

Now, let's go through the process of determining the root for a few examples and see how we can employ some of the aforementioned rules. As you'll see, determining the root is not always obvious without some knowledge of Arabic language grammars.

**Example 1 (اقتطع):** Here is a simple scenario: A word with five letters, also we know that the first and third letters belong to the extra letters group. So, we search the template dictionary for a template of length 5 where the first and third letters are (ا)

and (ت) respectively. Indeed we have the augmented verb form V (افتعل), knowing this, we can determine that the second, fourth and fifth letters form the root (قطع).

**Example 2 (انكسر):** Similar to Example 1, a word with five letters, also we know that the first and second letters belong to the extra letters group. So, we search the template dictionary for a template of length 5 where the first and second letters are (ا) and (ن) respectively. Indeed we have the augmented verb form VI (انفعل), knowing this, we can determine that the third, fourth and fifth letters form the root (كسر).

**Example 3 (معلومات):** A word with seven letters, a quick search in the template dictionary returns the template (مفعولات)(matching the template for derived noun passive participle for feminine/plural)

$$\text{مفعولات} = \text{م} + c_1 + \text{و} + c_2 + \text{ات} + c_3$$

So, we determine that the second, third and fifth letters form the root ( علم).

**Example 4 (الصيام):** The word consists of 6 letters, the first two letters matches the "definitive the" prefix (ال التعريف) also we know that the forth and fifth letters belong to the extra letters group. So, we search the template dictionary for a template of length 6 and starting with the "definitive the" prefix (ال التعريف). The search will return the template (فعال). However, the job is not quite done yet, the roots dictionary doesn't contain such a root ( صيم), therefore, we have one more step to do, that is the applying the normalization rules, in particular the rule that replaces the letter (ي) by ( و). Now we have the root (صوم) which is a valid root.

In the previous article (Article 1) we presented Arabic morphological analyzer **AMA**. The proposed stemmer can be implemented by extending the Arabic morphological analyzer **AMA** [AI12b, AI12a] by a fourth module "stemming module".

### 2.4.1 Stemming Module

This module will try to find all valid possible roots for a given word. First the module will search the template dictionary, using the *Masked Hamming Distance*, to find

and return a list of matching templates with the minimum *Masked Hamming Distance*. Then the module will normalize the list of roots by consulting the "Rules Engine Module" then accordingly the module will return a list of all the candidate roots for the input word. In Arabic when a word is formed (derived or inflicted) from a root. Some letters may be added, dropped, changed or replaced by other letters, during the normalization process we reverse these changes to its original state. Finally the stemming module returns the highest ranked root in the list of candidates or returns the entire list candidates, if requested.

## 2.5 Performance metrics

There are several criteria for judging stemmers: correctness, retrieval effectiveness, and performance. There are two error measurements in stemming algorithms, over-stemming and under-stemming. Over-stemming is an error where two separate inflected words are stemmed to the same root, but should not have been (a false positive). Under-stemming is an error where two separate inflected words should be stemmed to the same root, but are not (a false negative). Stemming algorithms attempt to minimize each type of error, although reducing one type can lead to increasing the other. It is possible to compare two separate stemming algorithms by comparing the output they produce. This provides a measure of the similarity (or conversely, the distance) between the two algorithms.

To conduct of our experiment. We have used data set taken from "KACSTAC" (King Abdulaziz City for Science and Technology Arabic Corpus) [KAC14]. "KACSTAC" consists of $869,800$ documents (total number of words is $732,780,509$ and total number of unique words $7,464,396$). This corpus is classified into: $200$ topics covering $20$ time period, from $28$ countries, collected from $10$ mediums (see [KAC14] for more information).

We have evaluated the performance of the new stemmer through extensive experiments. In order to assess our algorithm strength, first we conducted experiments based on [FF03], in their paper they listed the following measures to evaluate stemmers strength.

Suppose that $\mathcal{B}$ and $\mathcal{A}$ are the numbers of unique words before and after the stemming process respectively.

- W.C.C. is the Mean number of words per conflation class, $W.C.C. = \mathcal{A}/\mathcal{B}$.
- I.C.F. is the index compression factor, $I.C.F. = (\mathcal{B} - \mathcal{A})/\mathcal{B}$.
- W.C.F. is the word changing factor: the proportion of the words in the corpus that have been changed by the stemming process.

To get an idea of how the algorithm behaves in practice, we have implemented Algorithm REVENG and conducted a simple experimental study. A set of 10,000 words has been randomly selected from the set of unique keyword of our dataset (averaged over 10 runs) and counted the average values of W.C.C., I.C.F. and W.C.F. for each one of the three stemmers.

As is evident from the results Table 2.1, Khoja's stemmer generated the the least number of conflation classes with average of 3.82 words per class, the high average ratio is caused by over-stemming error and the conflation of unrelated words together in one class. The Light-10 stemmer, on the other hand, achieved an average 1.55 words per conflation class, hence the lowest index compression factor ratio (only 0.35) among the three stemmer.

Table 2.1 shows summary descriptive statistics for the results of testing the three stemmers on our dataset.

| Stemmer | B | A | W.C.C. | I.C.F. | W.C.F. |
|---|---|---|---|---|---|
| Light-10 | 10,000 | 6,428 | 1.55 | 0.35 | 91.97% |
| Khoja | 10,000 | 2,611 | 3.82 | 0.73 | 97.28% |
| **AMA** | 10,000 | 5,289 | 1.89 | 0.47 | 93.98% |

Table 2.1: **AMA** stemmer performance assessment

In the second experiment, we compared the new **AMA** stemmer to two superior stemmers, the root-based Khoja stemmer and the Light-10 stemming algorithm, the three stemmers were adopted as an indexing mechanism. The effect in increasing the F-measure was measured and compared between the three stemmers, using the Naive Bayes [Ris01, NEGH10] classifier in the evaluation. In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

$$\text{F-1} = \frac{precision \times recall}{precision + recall}$$

Precision and recall are the basic measures used in evaluating search strategies. Recall and precision measure the quality of your result. Recall is the ratio of the number of relevant records retrieved to the total number of relevant records in the database. Precision is the ratio of the number of relevant records retrieved to the total number of irrelevant and relevant records retrieved. Recall and Precision are usually expressed as a percentage.

$$\text{Recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|} = \frac{TP}{TP + FN}$$

$$\text{Precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|} = \frac{TP}{TP + FP}$$

Where TP (true positive) is a document which is relevant (positive) that was indeed returned (true), TN (true negative) is a document which is not relevant (negative) that was indeed NOT returned (true), FP (false positive) is a document which is not relevant but was returned and FN (false negative) is a document which is relevant but was not returned.

We run the experiment as follows: a data set of 100 document were randomly selected by randomly choosing medium, domain, time period and topic. The results, averaged over 10 trials. The experiments were run on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. We have implemented the new stemmer in $C\#$ language using Visual Studio 2010.

In Summary, Arabic information retrieval can be enhanced when the roots or stems are used in indexing and searching. Stemming reduces the vocabulary size by reducing

| Stemmer | Recall | Precision | F-1 Measure |
|---|---|---|---|
| Light-10 | 48.44% | 75.61% | 59.05% |
| Khoja | 71.88% | 45.10% | 55.42% |
| **AMA** | 65.38% | 72.34% | 68.69% |

Table 2.2: F1-measure comparison between Khoja stemmer, Light-10 stemmer and **AMA** stemmer.

variant words to a single form (stem or root), In our experiment, using Khoja's stemmer for indexing achieved an average F-1 measure of $55.42\%$ with low precision values $45.10\%$ caused by over-stemming and the conflation of unrelated words. The Light-10 stemmer, on the other hand, achieved an average F-1 measure of $59.05\%$ with a very low recall average $48.44\%$ therefore, it could not retrieve all the documents in many cases. The results, are illustrated in Table 2.2. In comparison with the other stemmers, the performance superiority of the proposed stemmer is clearly evident.

# Article: # 3

# Improved noisy channel model for Arabic spelling correction

We propose an expert system for Arabic spelling correction based on a generative Noisy Channel Model that goes beyond the primitive edit distance by presenting new conditioning factors, to compute the costs of a set of (learnable) string edit distance based on morphological characteristics and letters adjacency probabilities of Arabic words. The algorithm acquires the model parameters from a comprehensive training data-set, built for this purpose, consisting of pairs of erroneous and their correct words. The algorithm uses dynamic programming to calculate the edit distance/rules to learn the minimum total cost of transforming one string into another. Furthermore, we present the notion of "single candidate errors" and introduce a novel method for detecting and correcting many such errors that cannot be detected by currently existing techniques.

---

## 3.1 Background

Error correction and normalization generally are useful for a variety of tasks, including Text Authoring, Optical Character Recognition, Information Retrieval and Machine Translation. Algorithmic techniques for detecting and correcting spelling errors in text has a long history in computer science [BM00]. Fundamentally, a spell checker is made out of three components: An error detector that detects misspelled words, a candidate spellings generator that provides spelling suggestions for the detected errors, and an error corrector that chooses the best correction out of the list of candidate spellings. The majority of spell checking approaches can be thought of as calculating a distance between the misspelled words and each word in the dictionary. The shorter the distance, the higher the dictionary word is ranked as a good correction. The correction candidate set is selected by checking that each candidate satisfies the constraint that the edit distance between the word and its correction candidate is below certain thresholds, then only consider further those candidates that are sufficiently close to the best candidate. This can be done either by considering only the $n$-best candidates, or considering all those candidates whose frequency distribution analysis lie within a given value of the best candidate. Finally, the selection of the best correction strategy is based on the degree of confidence: The system will auto correct if it is very confident in the suggested correction, if it is confident only to a certain degree, then it produces the n-best corrections, otherwise, the system just highlights a word as a potential error without providing a correction. In general, spell checking techniques can be divided into these categories:

- Edit Distance Techniques [FW74, BM02].
- Phonetics Based Techniques [TM02, PZ83].
- Similarity Key Techniques [Dav62].
- N-Gram Based Techniques [Kuk92, DHK94, KDHT98].
- Probabilistic Techniques [CG91, KCG90].

Of course, these techniques are not completely independent from each other; rather they may have some overlaps.

In general, Spell checker can be divided based on the type of errors that can be detected and corrected into two broad categories: (1) *non-word errors*, words that cannot be found in a pre-compiled dictionary (a set of words typically lexicon or confusion set), such words are considered to be misspelled [ZPZ81] and (2) *real-word errors*, is also referred to as context sensitive spelling correction [MDM91], these are valid words in the dictionary but invalid with respect to their context.

The problem with generic methods is that they ignore important factors affecting the error patterns. Morphologically rich languages are characterized by a large number of morphemes in a single word, where morpheme boundaries are difficult to detect because they are fused together. At the same time all the methods that provide ranking mechanism by using a vast corpus and a language related training set, unfortunately, are not suitable for languages like Arabic because such corpora are not available so far, so these techniques cannot work efficiently alone while the rich morphological nature of such languages makes a morphology-based approach more suitable.

A morphology based spell checker has other advantages such as its ability to handle the name-identity problem, i.e., it can absorb new words and foreign words that are not included in the dictionary. New words may be absorbed by categorizing them into appropriate paradigms.

The fact that Arabic is a highly inflected language makes the correction of spelling errors extremely difficult because collecting all the possible word-forms in a lexicon is a relentless task.

The simplicity of inflection system in some languages, allows for reduced interest in research on morphological analysis in developing spell checking systems for such languages. In English, for example, the most common practice is to use a lexicon of all of the inflected forms with minimum set of morphological rules. That means, a great many language independent tools have been developed for syntactic and semantic analysis, the same cannot be said for morphological tools [AAA+92].

For the purpose of this work we use the the Arabic morphological analyzer **AMA** [AI12b, AI12a], to generate the segmentation rules (to be defined, shortly) and generate the training data set.

## 3.2 Prior Art

Several attempts was conducted to design error detection methods or to improve existing methods and investigate the effect of these methods on improving Arabic Information Retrieval. As a result, several versions of error detection methods have been developed. The naive (direct) way for detecting and correcting spelling errors is to match words in an input text against a list of correct words: if it is in the set it is correct, otherwise it is a mistake, such a words-list in Arabic can run into several millions [AI12a]. A space saver solution is to only store root-forms of words. That is, suffixes and/or prefixes are removed from the dictionary entries, an immediate problem with this approach is that its accuracy will depend very much on the stemmer used. The second problem, with such approach, is that the solution will be language specific since creating a version of this spell checker for another language would require a new stemming algorithm for that language or may not be possible depending on the language's morphology.

Another solution, adopted by [MS04a], is using a ternary search tree data structure for storing the dictionary, the solution combines the time efficiency of tries with the space efficiency of binary search trees, together they are faster than hashing for many typical search problems, and support a broad range of useful operations, like finding all keys having a given prefix, suffix, or infix, or finding those keys that closely match a given pattern. The proposed algorithm attempts to select the best choice among all possible corrections for a misspelled term using word frequency counts as a popularity ranking, together with other information such as meta-phone keys.

Shaalan et. al. [SAG03] introduced a tool that is capable of recognizing and suggesting correction of ill-formed input for common spelling errors. It is composed basically of Arabic morphological analyzer, lexicon, spelling checker, and spelling corrector.

Rytting et. al. [RRB+10] employed a modular approach, developing separate modules for mistypings, phonetic confusions, and other dialectal confusions, each modelled through a weighted finite state transducer (FST). The resulting FSTs are composed with a finite state machine. Accepting all strings corresponding to entries in an electronic dictionary. The composed finite state transducer calculates the best paths yielding unique, valid strings, i.e., the dictionary entries most likely to have been the

intended query given the misheard, transliterated, or mistyped input text.

In dictionary based methods, we have to deal with another problem, namely, "out-of-vocabulary" problem or "false-positives" (a word marked as a mistake when in fact it is correct), false-positives are words such as proper nouns, special domain terms or foreign words borrowed from other languages, such words are not found in traditional dictionaries. Spelling mistakes involving valid out-of-vocabulary words are uncorrectable. The system will either make false-corrections or have no suggestions at all.

In order to minimize the out-of-vocabulary problem a comprehensive word list covers all domains have to be used (if such list can be collected), So far so good, however this will cause in some mistakes to be considered as correct words. These are called "false-negatives" (a mistake that is judged to be correct) –also called a word-to-word mistake, false-negatives will increase as a result of a including many rare words in the dictionary since the mistake also happens to be a dictionary word.

An $q$-gram model is a sequence of $q$ adjacent letters in a word. The more $q$-grams two strings share the more similar they are.

$$\text{similarity coefficient } \delta = \frac{\text{number of common } q\text{-grams}}{\text{total number of } q\text{-grams}}$$

A formal definition of the $n$-gram and the $n$-gram similarity is as follows: For a given string $\boldsymbol{s}$, of length $n$, drown over a finite alphabet $\Sigma$ and $q$ is a positive integer number, a $q$-gram of $\boldsymbol{s}$ is a pair $(g, i)$, where $g$ is the $q$-gram of $\boldsymbol{s}$ starting at the $i$-th position, that is $g_i = \boldsymbol{s}[i\,..\,i+q-1]$, the set of $q$-grams of $\boldsymbol{s}$, denoted by $G(\boldsymbol{s}, q)$, is obtained by sliding a window of length $q$ over the the string $\boldsymbol{s}$. There are total of $|\boldsymbol{s}| - q + 1$ $q$-grams in $G(\boldsymbol{s}, q)$.

The $q$-gram similarity of two strings is the number of $q$-grams shared by the strings, which is based on the following lemma.

**Lemma 3.2.1 (The $q$-gram lemma)** *[Ukk92] Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be strings with the edit distance $\delta(\boldsymbol{x}, \boldsymbol{y})$. Then, the $q$-gram similarity of $\boldsymbol{x}$ and $\boldsymbol{y}$ is at least $Q_{sim} = max(|\boldsymbol{x}|, |\boldsymbol{y}|) - q + 1 - (q \times \delta(\boldsymbol{x},\ \boldsymbol{y}))$. Given strings $\boldsymbol{x}$ and $\boldsymbol{y}$, let an occurrence of $x[1\,..\,i]$ with at most $k$ differences end at position $j$ in $\boldsymbol{y}$. Then at least $i + 1 - (k + 1) \times q$ of the $q$-grams in $x[1\,..\,i]$ occur in the substring $\boldsymbol{y}[j - i + 1\,..\,j]$.*

$Q$-gram technique doesn't exhibit good performance on short words. For example when using tri-grams, the words of length 3 will share no tri-gram between themselves if they contain a single-character error. $q$-gram similarity measure works best for insertion and deletion errors, well for substitution errors, but very poor for transposition errors. The use of $q$-gram model instead of a dictionary based is a very space-efficient method for detecting mistakes. Simply, it works by looking for words with unusual character sequences. Therefore, $q$-gram is by no means accurate and its generally considered inadequate for spelling correction. Especially, due to the high graphemic nature (all of the letters and letter combinations that represent a phonem) of Arabic words. It is more often used in Optical Character Recognition (OCR), where errors are more likely to result in unusual letter sequences. A more refined method is detection through language modelling, this method has been used frequently for the purpose of spelling correction. Based on Markov chain [Mar13], Markov assumption, that is the future behaviour of a dynamical system only depends on its recent history. In particular, in the $k$-th order Markov model (Equation. 3.1), the next state only depends on the $k$ most recent states, by calculating probabilities of each $k$-th-gram and computing the probability of the word with some threshold for when an improbable word is judged as a mistake.

Assuming that we have a sequence $s = w_1, w_2, \cdots w_n$ of random variables (more specifically, sequence of words in the case of language modelling) then the probability of producing the sequence is

$$\begin{aligned} P(w_1^n) &= P(w_1, w_2, \cdots w_n) \\ P(w_1^n) &= P(w_1)P(w_2|w_1)P(w_3|w_1^2)\cdots P(w_n|w_1^{n-1}) \\ P(w_1^n) &= \prod_{i=1}^{n} P(w_i|w_1^{i-1}) \\ P_k(w_1^n) &= \prod_{1}^{n} P(w_i|w_{i-k}^{i}) \end{aligned} \tag{3.1}$$

However, the observations in [AI14] suggested that Arabic language suffer from data sparseness which usually leads to high rate of false-positive and false-negative errors. Also the set of experiments conducted by Zribi and Ahmed in [ZA03] showed

that the average number of forms that are lexically close for Arabic language (without vowel marks) is 26.5. Thus, Arabic words would be much more closely than French and English words (3 for English and 3.5 for French), such phenomenas make the detection of errors using language modelling performance to be unsatisfactory.

A hybrid approach, introduced in [HY07], by utilizing morphological knowledge in form of consisting root-pattern relationships, and some morpho-syntactical knowledge based on affixation and morpholo-graphemic rules, to deliver the word recognition and non-word correction process. Then, based on probabilistic measures, the system complete the task of the correction by locating, reducing and ranking of the most probable correction candidate for Arabic words.

A system of analysis of Arabic texts based on the approach of multi-agents was introduced in [AB12]. It consists of a set of agents, using a direct communication by sending messages. These agents work together in order to make syntaxes' analysis of a sentence, given by the user, by determining its syntax composition. The major drawback of such system is the time taken by the agents for communication and interaction.

Arabic "GramCheck", introduced in [Sha05], is another syntax-based grammar checker for modern standard Arabic. The system is based on deep syntactic analysis and relies on a feature relaxation approach for detection of ill-formed Arabic sentences.

Shaalan et. al. [SAP+12] presented a context-independent spelling correction tool using a finite-state automaton that measures the edit distance between input words and candidate corrections and the noisy channel model and knowledge-based rules for scoring.

### 3.2.1 Noisy Channel Model

The concept behind the noisy channel model [Sha48] is to consider the process that is causing a misspelling as a noisy signal which has been distorted in some way during communication. Based on this assumption, it is then straightforward to deduce the actual correction, if one could identify how the original word was distorted.

Typically, a language model (source model) is used to capture contextual information, while an error model (channel model) is considered to be context free in that it doesn't take into account any contextual information in modelling word transformation probabilities. This approach was first employed for spell checking [KCG90].

$$Pr(t|c) = \begin{cases} \frac{del[c_{p-1}, c_p]}{chars[c_{p-1}, c_p]} & \text{if deletion} \\ \frac{add[c_{p-1}, t_p]}{chars[c_{p-1}]} & \text{if substitution} \\ \frac{sub[t_p, c_p]}{chars[c_p]} & \text{if insertion} \\ \frac{trn[c_p, c_{p+1}]}{chars[c_p, c_{p+1}]} & \text{if transposition} \end{cases} \tag{3.2}$$

The intended correction, $c$, can often be recovered from the typo, $t$, by finding the correction $c$ that maximizes $Pr(c)Pr(t|c)$, where first factor, $Pr(c)$, is the prior model of word probabilities; the second factor, $Pr(t|c)$, is the noisy channel model that accounts for spelling transformations (insertions, deletions, substitutions and transpositions) on letter sequences, where $c_p$ is the $p$-th character of $c$, and likewise $t_p$ is the $p$-th character of $t$.

The model formalizes the task of selecting the most likely candidate as an instance of Bayesian Inference [Bay63]. Probabilities are estimated from these matrices by dividing by $chars[x, y]$ and $chars[x]$, the number of times that $xy$ and $x$ appeared in the training set, respectively. At heart, the Bayesian model is a probabilistic model based on statistical assumptions which employs two types of probabilities: the prior probability $P(S)$ and the likelihood probability$P(\hat{S}|S)$ which can be calculated using Bayes' Rule. All the influences of word frequency, context, and word similarity are required to be computed so that they can be evaluated quantitatively on a probabilistic scale and can be easily combined.

Formally, we wish to find the intended $S^*$ that has the highest likelihood given the observed sentence $\hat{S}$:

$$S^* = \arg\max_{S \in V} P(S|\hat{S})$$

Applying Bayes' Rule.

$$P(S|\hat{S}) = \frac{P(\hat{S}|S) \times P(S)}{P(\hat{S})}$$

$$S^* = \arg\max_{S \in V} \frac{P(\hat{S}|S) \times P(S)}{P(\hat{S})}$$

And dropping the constant denominator.

$$S^* = \arg\max_{S \in V} P(\hat{S}|S) \times P(S)$$

The term $P(S)$ is the language model and $P(\hat{S}|S)$ is the error model, where $S^*$ is the best estimate of $S$. In order to to overcome the sparse data problem, the system have to assign non-zero probabilities to unseen words, to improve the accuracy of the estimated language model, we have to shrink the probabilities of the observed words in the vocabulary so that it will distribute the borrowed probabilities value to the unseen words.

$$P(word) = \frac{occ(word) + k}{N + k \times V} \tag{3.3}$$

Where $occ(w)$ is the number of time the word $w$ occurs in the data set, $N$ is the number to tokens in the data set, $k$ is the smoothing parameter and $V$ is the size of the vocabulary (number to token types). A simple technique called the additive smoothing, which gives all the unseen words equal probabilities. A better smoothing method should give different words different probabilities. Notice that the noisy channel model offers the possibility of correcting misspellings without a dictionary, as long as sufficient data is available to estimate the source model factors.

## 3.3 Our Approach

For the noise channel model, we need some way to learn the parameters for the spelling mistakes probabilities. Using this model, we can find the highest likelihood error-free suggestions for an observed word by tracing all possible paths from the language model through the noise model and ending in the observed word as output. For our noise model, we created a weighted finite state transducer which accepts error-free input, and outputs erroneous sentences with a predefined error types and probability. To model various types of human errors, we created several different noisy models and merge them together, resulting in a layered noise model. Beside the dictionary of

inflected/derived forms, the morphological analyzer makes use of a set of dictionaries that contain all affixes and applies a set of predefined rules of all possible partitions in prefixes, stems and suffixes. These dictionaries and grammatical rules shall be used in the process of segmentation of words and the generation of errors as described in Section 3.8.

## 3.4 Error Categories in Arabic

In classifying the spelling errors, there is a primary distinction between "spelling mistakes" and "typing errors". Spelling mistakes (or competence errors) are caused when the author genuinely believes that a word is spelt in a particular incorrect way. Spelling mistakes are systematic errors that are due to language influence. Within the category of spelling mistakes, we distinguished errors according to language influence (phonological, morphological, lexical and orthographic), for instance, phonological spelling mistakes often involve phonological proximity, particularly, with vowels being substituted for each other.

In contrast, typing errors (performance errors) are caused when the author intends to type a particular word of key-strokes and fails. Typing errors are, random, unsystematic errors that capture the specific deviation from the target spelling. For instance, the typist try to hit the letter "a" but instead hit the adjacent letter in the keyboard "s". Typing errors were subdivided into single and multiple letters violations (additions, deletions, substitutions, and transpositions), and word boundary violations.

Furthermore, depending on the source of the examined text, the errors may be dominated by one type. For example, hand-writing recognition would be free from typing errors while a text generated by OCR applications usually contains mistaking characters for visually similar ones, often unlike human errors (e.g. mistaking "m" for "iii").

Its not an easy task to assign a single group to a certain spelling error. Having said that, in our approach, we analyzed and classified the spelling errors based on the best method of correction rather than grouping them based on the cause or source of the error.

In the following, we summarize six general type of spelling errors:

- Phonetic mistakes: where the mistake sounds similar to the intended word (e.g., [حياة → حيات] and [لكن → لاكن]).
- Typographic Errors: resulting form pressing the wrong character (e.g., the letter to the right or to the left) on the keyboard ([باب → ياب] and [خياط → حياط]).
- Word boundary Error: (1) Run-on Error (missing a space between adjacent words) (e.g., non-visible [ينتظر طويلا → ينتظرطويلا] or visible [آثارهم بهم → آثارهم ربهم]). (2) Splitting Errors (extra a space) (e.g., non-visible [اعتر اف → اعتراف] or visible [الحقيقية → الحق يقية]).
- Morphological mistakes: resulting from the two main processes of word formation (inflection and derivation) (e.g., [مهتدون → مهتديون] and [انكسر → إنكسر]).
- Orthographical mistakes: such as the confusion of the different type of the glottal stop letter Hamzaa (الهَمْزة) (e.g., [ضوئك or ضوءك]).
- Grammatical mistakes: Such as wrong gender, person, number, voice or tense (e.g., [الطلاب كتبوا الدرس → الطلاب كتب الدرس] and [هذا الفتاة → هذه الفتاة]).

Note that existing spell checkers are more successful in handling performance errors than competence errors [RH05]. This finding extends our findings of single-error words which showed that competence errors are generally harder to correct than performance errors.

## 3.5 String Similarity measurement

Similar to [CHO05a], most of the systems reviewed in this study we focus on two classical and well established string matching algorithms to detect misspelled terms, namely the edit distance and the longest common subsequence algorithms. The results, presented in [CHO05a], showed that both algorithms can correct misspelled terms but the degree of success varied. The edit distance was found to be more effective than the longest common subsequence with most of the tested type of errors. Therefore, the longest common subsequence approach will not be adapted in this study.

Recall that the edit distance between two strings is the distance $\delta(\boldsymbol{x}, \boldsymbol{y})$ between two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ is the minimal cost of a sequence of operations that transform $\boldsymbol{x}$ into $\boldsymbol{y}$ (and $\infty$ if no such sequence exists).

**Definition 41 (Spelling Correction)** *The following definition from [BM00]: Given an alphabet $\Sigma$, a dictionary $D$ consisting of strings in $\Sigma^*$ and a string $s$, where $s \notin D$ and $s \in \Sigma^*$, find the word $w \in D$ that is most likely to have been erroneously input as $s$. The requirement that $s \notin D$ can be dropped, but it only makes sense to do so in the context of a sufficiently powerful language model.*

### 3.5.1 Learnable Edit Distance

[RY98] demonstrates the improvements obtained using their method over Levenshtein distance [Lev66] on the task of matching natural language strings and their different edit operations have varying significance in different domains. In their model, a string alignment is equivalent to a sequence of character pairs generated by edit operations emitted by a hidden Markov model with a single non-terminal state. [RY98] defined the stochastic edit distance between two strings $\boldsymbol{x}$ and $\boldsymbol{y}$ to be:

$$\delta(i,j) = \min \left\{ \begin{array}{l} \delta(i-1, j-1), \text{ if } x_i = y_j \\ \delta(i-1, j-1) + substitution[x_i, y_i], \text{ if } x_i \neq y_j \\ \delta(i-1, j) + insertion[x_i] \\ \delta(i, j-1) + deletion[y_j] \end{array} \right\} \tag{3.4}$$

[CG91] introduced a new error model, allowing generic string-to-string edits rather than single character, two improvements are made. First, instead of weighing all edits equally, each unique edit has a probability associated with it. Second, insertion and deletion probabilities are conditioned on context. It proved advantageous to model substitutions of up to 5-letter sequences.

Positional information is a powerful conditioning feature for rich edit operations. [TM02] explicitly build a separate error model for phonetic errors, using the pronunciations of the correct words and the estimated pronunciations of the misspellings to learn phone-sequence-to-phone-sequence edits and estimate their probabilities, and build an error model for letter strings using [TM02] learning algorithm, to train these two mod-

els on the same data set of misspellings and correct words then combine the two models to estimate scores as follows:

$$Score = \log P_{LTR}(w|r) + \lambda \log P_{PHN}(w|r)$$

Where the probability distribution of letter-based model over words is $P_{LTR}$ and the probability distribution of phone-based model over pronunciations is $P_{PHN}$.

Given an alphabet of symbols $\Sigma^* = \Sigma \cup \{\epsilon\}$, the full set of edit operations is $E = E_s \cup E_d \cup E_i$, where $E_s = \{(a;b)|a;b \in \Sigma\}$ is the set of all substitution and matching operations $(a;b)$ and $E_i = \{(\epsilon;a)|a \in \Sigma\}$ and $E_d = \{(a;\epsilon)|a \in \Sigma\}$ are sets of insertion and deletion operations respectively. Rather than simply counting the number of required edit operations to change $\boldsymbol{x}$ of length $|\boldsymbol{x}|$ into $\boldsymbol{y}$ of length $|\boldsymbol{y}|$, the additive value 1 in Formula. 1.2 can be replaced by plugging an edit cost function in an edit distance algorithm, $ins[x(i)]$, $del[x(i)]$, and $sub[x_i, y_j]$, that takes into account the nature of the symbols $x_i, y_j \in \Sigma^*$ and the position in the word involved in the edit operation, where $\Sigma$ is the alphabet. In this case, the edit distance between $\boldsymbol{x}$ and $\boldsymbol{y}$ becomes the minimum cost of all sequences of edit operations which transform $\boldsymbol{x}$ into $\boldsymbol{y}$.

Bilenko and Mooney [BM02] presented an analogous generative model can be constructed for string distance with affine gaps, the gap penalty is calculated using the affine model: $cost(g) = s + e \times \ell$, where $s$ is the cost of opening a gap, $e$ is the cost of extending a gap, and $\ell$ is the length of a gap in the alignment of two strings, assuming that all characters have a unit cost.

The noisy channel can be modelled using a simple probabilistic finite state transducer or a Markov chain with no input memory [BJ06] modelling each edit operation corresponds to the probability of observing a pair of characters, or a single inserted/deleted character in the alignment. A string alignment is equivalent to a sequence of character pairs generated by edit operations emitted by a memory-less stochastic transducer. Each edit operation corresponds to the probability of producing a substitution $p(x_i, y_j)$, an insertion $p(\epsilon, y_j)$, or a deletion $p(x_i, \epsilon)$, where probabilities of all operations are normalized. [OTAT08] presented a discriminative approach for generating candidates for string transformation modelled by logistic regression build a binary classifier that, when given a source string s, decides whether a candidate $t$

should be included in the candidate set or not.

## 3.6 Building Aho-Corasick Automata

Simply, we are interested in searching for all occurrences of all patterns taken from a finite set of patterns in a given text. Trivial approach (one pattern at a time), Imagine using a linear time method (e.g. Knuth-Morris-Pratt) that runs in $O(n + m)$ time, where $n$ is the size of a single preprocessed pattern that we compare to a text of size $m$. An immediate solution to speed this up is to perform all comparisons simultaneously against a structure that contains all the patterns. Keyword trees help us do exactly this and can be thought of as a compressed set of patterns that we compare to the text: The solution, in [AC75], was by creating a finite state machine to match the patterns with the text in one pass.

The Aho-Corasick Automaton [AC75] for a given finite set $P$ of patterns is a deterministic finite automaton $A$ accepting the sets of all words containing a word of $P$ as a suffix. $A = (Q, \Sigma, g, f, q_0, F)$, where function $Q$ is the set of states, $\Sigma$ is the alphabet, $g$ is the forward transition, $f$ is the failure link i.e. $f(q_i) = q_j$, if and only if $S_j$ is the longest suffix of $S_i$ that is also a prefix of any pattern, $q_0$ is the initial state and $F$ is the set of final (terminal) states. The construction of the AC automaton can be done in $O(d)$-time and space complexity, where $d$ is the size of the dictionary.

## 3.7 Quick Match

There are however a number of situations where the noisy model is insufficiently general. At this point, we address the related question of selecting an appropriate structure with just the right amount of trainable parameters. Hence, the Quick Match is the component that we use to identify possible correct spellings of predefined error and we already know the exact correct word for that error. Therefore, we present the notion of single candidate errors and introduce and novel method for detecting and correcting many such errors that cannot be detected by a conventional techniques.

The Quick match component is based on comprehensive dictionaries of morphological templates/rules and deep phonological/orthographical alterations rules such as

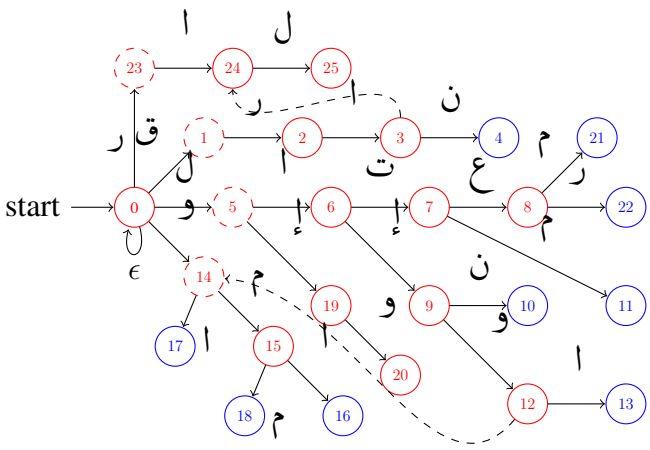



Figure 3.1: Building Keywords tree from errors dictionary

gemination, vowelization and substitution, etc. The Quick match components, for detection of misspelled Arabic words, is capable of detecting and correcting of certain widely observed errors, such errors that cannot be detected by a conventional techniques. When analyzing such error, we notice that they all share a common property that is they all have a single candidate suggestion. We use a keywords tree to model these errors and store them in as a dictionary, each error identified as a single candidate error will be added to the quick match dictionary. The quick match dictionary also contains a set of the most common Arabic particles such as conjunctions, pronouns, demonstratives, prepositions, conditional, exclusion, determination and negation letter, these sets of words will be used as pairs of erroneous and their correct words to build the error dictionary.

## 3.8 Improved noisy channel model

Our model works by learning tailored string to string transformations with the use of the language morphological characteristics, along with the probabilities of each of these transformations. In this case the model accounts for both typing errors or spelling mistakes. This more powerful model gives significant improvements in accuracy, as shown in the results (Table 3.1), over conventional approaches (such as single

error, generic, or phonic distances) to noisy channel spelling correction. Separate error modules for keyboard mistypings, phonetic confusions, and dialectal confusions are combined to create a weighted finite-state transducer that calculates the likelihood that an input string could correspond to each suggestion candidates set. Results are ranked by the estimated likelihood that a candidate could be misspelled or mistyped for the given word. For typing errors, we allowed all spelling errors which were a Damerau-Levenshtein distance of ([Dam64]; [Lev66]). This is as previously discussed rather similar to the standard noisy channel model for spelling correction [BM00].

Typically, for word length $\ell$, we have $\ell$ deletion errors, $36 \times \ell$ substitution errors (29 unique letters in Arabic, 36 letter shapes), $\ell - 1$ transposition errors, and $36 \times (\ell + 1)$ insertion errors. Note that these numbers are derived from the possible number of edit operations (deletion, substitution, transposition and insertion) that can be performed on a given word of length $\ell$.Also we set the maximum word length $\ell$ to 22.

We use trainable transducers to model spelling mistakes, we specified a set of $n$ (total number of possible edit operations) parameters for each possible word length $\ell$. These parameters represent the total probability of making a spelling error for a given word length. Then, for each word length $\ell$ we distribute the probability of each possible spelling error equally. After that, we manipulate these probabilities based on the various conditions, these conditions are determined based on (1) comprehensive dictionaries of morphological templates/rules and deep phonological/orthographical alterations rules and (2) intensive analysis of the previously seen error in Arabic text. Special considerations are made for each type of errors such as word boundaries errors, run-on and splitting errors on the preceding and the following disconnected letters:

(ة ؤ ئ إ أ ء ا ؤ و ز ر ذ د)

For example, considering the cases of an extra or missing white space following the disconnected letters should assign a higher weight than the rest of letters.

We are not aware of any Arabic spell checker that can detect this type of error, especially, considering the effects where white space is a factor in the edit distance.

We also condition on lexicon-based features, which are generated by checking whether a given word and each of its correction candidates is in a relevant spelling lexicon.

In order to allow the model to take in the consideration the position of the error, we segment the input words according to the morphological derivation/ inflection rules by defining two regions as follows: given an input word $w$ each character in $w$ either belong to the lexicon region or to the affix region. The lexicon region is the 3 or 4 letters in the case of trilateral or quadrilateral root/stem respectively while the affixation region is the letters that will be added to the stem (as prefix, infix or suffix) to construct the new form of the word according to the morphological segmentation rules [AI12a], for example:

- [استفعل → فعل + است],
- [ينفعل → فعل + ين],
- [يفتعل → عل + ت + ف + ي],
- [افعول → ل + و + فع + ا].

Since transitions are described by a function [AI12a], the function $\delta$ specifies the given word as a sequence of lexicon and affixation regions.

We investigated several, practical and theoretical, aspects of natural language to learn statistics about words, lemmas and stems frequencies as well as letters distribution and relationships between word length and letters adjacency distribution, Based on our findings we condition on the letters and affixes combination and neighbouring statistics. By finding any irregular combination of letter that don't follow each other in a certain word.

Next, we allow the process to misspell the list of correct words generated using ***AMA*** morphological analyzer. The task of generating pairs of misspelled and correct words was done on total of over four million words, then encoded as a weighted finite-state transducer.

## 3.9 Candidate Ranking

The "Morphology confusion matrix" $M$ is a $N \times N$ matrix, where $N$ is the number of segments in the morphology set. The entry $M[i, j]$ represents the number of times the

the $i$-th segment has been substituted by the $j$-th segment in the training set, as a factor of the total number of total occurrences of the $i$-th segment. A straightforward method to calculate the substitution penalty from the morphology confusion matrix would be to directly use the value $1 - M[i, j]$ as the substitution penalty for the pair of segments $i$ and $j$. However, experiments showed that the penalties obtained in this way were very close to each other and sometimes congested. Therefore, the substitution penalty for the segments pair $i$ and $j$ will be calculated as follows:

$$S[i, j] = \log((M[i, i]/M[i, j]); i \neq j \tag{3.5}$$

Since in most cases $M[i, i] >> M[i, j]$. Therefore, the $\log$ conversion is applied to keep the substitution penalties within ranges. The experiment showed that the penalty function in Equation 3.5 provides enough variance among the substitution penalties for various segment pairs as compared to directly using the values $1 - M[i, j]$. It must be noted also that $M[i, j] \neq M[j, i]$

For practical purposes, the substitution penalty for only the top $M$ confusing segments may be determined while for the rest, substitution may be forbidden by setting an infinite penalty (value of $-1$). All counts were smoothed using Laplace smoothing Equation 3.3, $0.5$ is added to every count, and the effective corpus size is increased appropriately.

In a nutshell, the word segmentation process occurs during the morphological analysis, then building the set of pairs of correct and misspelled words and stored as a weighted finite-state transducer, the correction process is carried out by finding the sets of segments that most similar to the input word using the learnable edit distance then selecting the top candidates is done by computing the best set of segments among the set of suggested segments (this will be determined using the computed probabilities of the these sets).

## 3.10 Experiment

### 3.10.1 The Corpora

For this project, we used two data sets to conduct our experiment, the first data set (testing data set) KACSTAC[1] (King Abdulaziz City for Science and Technology Arabic Corpus) [KAC14], KACSTAC consists of $869,800$ documents with $732,780,509$ total number of words (tokens) and $7,464,396$ total number of unique words (types), the second data set, (training data set) of $260,922,024$ words, is generated by the Arabic Morphological Analyzer ***AMA*** [AI12a], the data set contains $58,463,856$ verbs, $128,837,016$ derived nouns and $73,621,152$ verbal nouns.

### 3.10.2 Performance metrics

We compared our system with famous Hunspell [Hun13] spell checker, Hunspell is the spell checker of LibreOffice, OpenOffice.org, Mozilla Firefox and Thunderbird, Google Chrome, and it is also used by proprietary software packages, like Mac OS X, InDesign, memoQ, Opera and SDL Trados.

In Table 3.1 Best-1, Best-5 and Best-10 accuracy represent the percentage of time the correct answer is one of the top 1, 5 and 10 answers returned by the system.

We believe this to be the first spelling correction system designed, specifically for Arabic, for detecting and correcting such errors that usually go undetected using existing spell checkers. To evaluate our spell checker, a series of experiments is carried out, based on on both real errors found in the testing data set (the system hasn't seen these error before) and artificial errors created using the training data set (data used for training the system), that shows that our novel weighted edit distance not only significantly outperforms the baseline edit distance but also it has been shown to be superior to the widely well known Hunspell spell checker as shown in Table 3.1.

[1] See http://www.kacstac.org.sa/ for more information

| Experiment Results | | | | |
|---|---|---|---|---|
| Baseline edit distance | | | | |
| Experiment | Precision | Recall | Accuracy | F-measure |
| Best-10 | 45.90% | 71.79% | 56% | 56% |
| Best-5 | 38.09% | 63.15% | 47% | 47.52% |
| Best-1 | 25.75% | 44.73% | 30% | 32.69% |
| Hunspell spell checker | | | | |
| Experiment | Precision | Recall | Accuracy | F-measure |
| Best-10 | 91.66% | 88% | 90% | 89.79% |
| Best-5 | 81.63% | 80% | 81% | 80.80% |
| Best-1 | 56% | 70% | 65% | 62.22% |
| Improved noisy channel | | | | |
| Experiment | Precision | Recall | Accuracy | F-measure |
| Best-10 | 92.30% | 90% | 93% | 91.13% |
| Best-5 | 88.09% | 92.5% | 92% | 90.24% |
| Best-1 | 83.72% | 90% | 89% | 86.74% |

Table 3.1: Spelling Correction Evaluation Results

# Article: # 4

# Degenerate Finite State Automata for Arabic Morphology

In this article, we propose DeFSA, a Degenerate Finite State Automaton, to facilitate the expression of various non-concatenative morphological phenomena of the Arabic language in an efficient way. DeFSA is a lightweight data structure which is able to validate Arabic words efficiently. The DeFSA augments regular FSA with finite memory (auxiliary data structure) in a restricted way that saves the overall space but does not add expressivity. We further implement DeFSA with the help of a celebrated data structure in Stringology called *Directed Acyclic Word Graph (DAWG)*. The experimental results show that DeFSA and its DAWG implementation are much more efficient than their existing alternatives (e.g., regular FSA).

---

## 4.1 Introduction

*Morphology* is the domain of linguistics that studies the formation of words. It is traditional to distinguish the *surface form* (i.e., the actual word itself as found in the text) of a word from its *lexical forms* or *lemma*. The latter typically consists of the canonical dictionary citation form of the word together with the terms that convey the list of features and morphological properties of that particular form. In the study of linguistics, *Words* are typically assumed to be composed of the smallest meaningful units called *morphemes*, the basic building blocks in morphology. In fact, a morpheme can be seen as the minimal unit of grammatical analysis. A *root*, simply consisting of a single *morpheme*, is a basis for compounding and affixation. A *stem* is a form of a word, made up of one or more morphemes, before any inflectional affixes are added: *affixes* are word elements attached to words that may either precede as *prefixes*, follow as *suffixes* or get inserted inside a word *stem* as *infixes*. The study of word formation is about the grammars that govern the combinations of *stems*, *affixes*, and other types of *morphemes*.

Word formation and morphological alternations are the two central problems in the study of morphology. Finite-state morphology is an attempt to account for these phenomena within the context of regular sets and regular relations. Although the two problems are certainly related, the solution for one in the finite-state domain does not automatically imply that the other is also solved. The problem of morphological alternations arises from the fact that a morpheme may have alternate realizations depending on its phonological environment and the composition of the word. We concentrate in this paper on morphological alternations, and put aside the question of word formation.

## 4.2 Preliminaries

A *degenerate string* $x = x[1..n]$ on an alphabet $\Sigma$ is a sequence of nonempty subsets of $\Sigma$. We say that $x[i_1]$ and $x[i_2]$ **match** (written $x[i_1] \approx x[i_2]$) if and only if $x[i_1] \cap x[i_2] \neq \emptyset$. So, a position of a degenerate string may match more than one element from $\Sigma$; such a position is said to have a *non-solid* symbol (also called a character class). If in a position we have only one element of $\Sigma$, then we refer to this position as *solid*. The definition of length for degenerate strings is the same as for regular strings: a

degenerate string $\mathcal{S}$ has length $n$, when $\mathcal{S}$ has $n$ positions, where each position can be either solid or non-solid.

In language processing, an FSA can be used to recognize or generate a specific language defined by all possible combinations of characters (conditional labels) on each of the edges generated by traversing the FSA from the initial state to the end state.

Each path from the initial state to a final state can be seen as a mapping between a surface form and its lexical form (lemma).

Since we will be using a celebrated data structure in Stringology, namely, the Directed Acyclic Word Graph (DAWG), in the rest of this section we briefly review it.

### 4.2.1 Directed Acyclic Word Graph (DAWG)

The DAWG[1] is a deterministic finite automaton representing a set of words in which each edge is labeled with a character. The characters along a path from the initial state to a node form a substring that the node represents. A DAWG is an extremely useful tool to search through a large lexicon. An important advantage of the DAWG representation of the lexicon is its compactness; this permits extremely fast word searching while keeping the entire lexicon in the primary memory and thereby minimizing costly disk accesses [AJ88]. DAWGs have been used in many applications in NLP including constructing dictionaries and transducers for spell-checking, morphological analysis, two-level morphology, restoration of diacritics, perfect hashing, and document indexing [DMWW00]. A DAWG containing all the words in a dictionary can very quickly answer the question of whether or not an input word is in that dictionary.

However, a DAWG cannot support the capability of providing suggestions for alternate spellings for a given word since it does not contain any values associated with the words. There exists however data structures that support inclusion of associated values. One such data structure is the "Minimal Acyclic Finite State Transducers" (see, [MM01, DMWW00]).

There is also a variant of the DAWG called a compact DAWG (CDAWG) that further reduces the memory footprint of the structure. CDAWG can be constructed from DAWG by removing nodes that have only one outgoing edge, and represent-

[1] Also referred to as the *Minimal Acyclic Finite State Automaton* in the literature.

ing the deleted paths between remaining nodes with edges that are labeled with the path's label. CDAWG can be obtained from DAWG in linear time (for more details see [MS04b]). In [HC03] an efficient implementation of CDAWG was proposed. While the previous implementations of CDAWG had required from $7n$ to $23n$ bytes of memory space, this implementation required $1.7n$ to $5n$ bytes for a text $T$ of length $n$.

## 4.3 Arabic Morphological analyzer *AMA*

In this section we briefly review the Arabic morphological analyzer ***AMA***, presented in the first article (Article 1) of this chapter.

***AMA*** is basically a finite state transducer (FST), was obtained by directly implementing a comprehensive list of Arabic roots, a dictionary of Arabic morphological patterns and a set of Phonological/Orthographical alternations rules.

***AMA*** has many functions. It can validate a word (accept or reject) and also can generate different forms of a given word. It also can be used as a stemmer: give it a word as input and ***AMA*** will return the stem of that word. Also ***AMA*** can return all the word annotations such as morphological properties (category, type, part of speech tags) of the input words. Another way to use ***AMA*** is to give it the annotations as input and ***AMA*** will return all the words that match these annotations. However, the main issue with ***AMA*** is that its very slow when matching words and the list of words is too big in size.

In this article, we propose `DeFSA`, a finite state machine to validate Arabic words that achieves a significant reduction in space requirements as compared to the regular finite state automaton constructed for the same purpose. In `DeFSA`, excessive duplication of states and edges are cleverly avoided. This is done by employing the concept of degenerate strings (to be defined shortly) to maintain useful templates rather than actual words and by constructing the automaton based on the former (i.e., templates) rather than the latter (i.e., actual words). As a result, `DeFSA`, provides an elegant solution to overcome the size and speed issue of `AMA`. However, `DeFSA` cannot mimic all the functionalities of `AMA`; it can only be used for validating Arabic words.

We note here that, our approach is not only limited to the Arabic Language; rather it is supposed to work for any language that follows specific morphological rules in the word generation process and can be represented as a set of morphological patterns

or regular expressions. However, as has already been pointed out above that we are motivated to work on the Arabic language and hence our experiments are tailor made for Arabic language.

We further implement DeFSA with the help of a celebrated data structure in Stringology called Directed Acyclic Word Graph (DAWG). As will be reported later, DeFSA performs much better than the regular FSA in terms of speed and memory. Finally, the implementation of DeFSA with DAWG turns out to be even better.

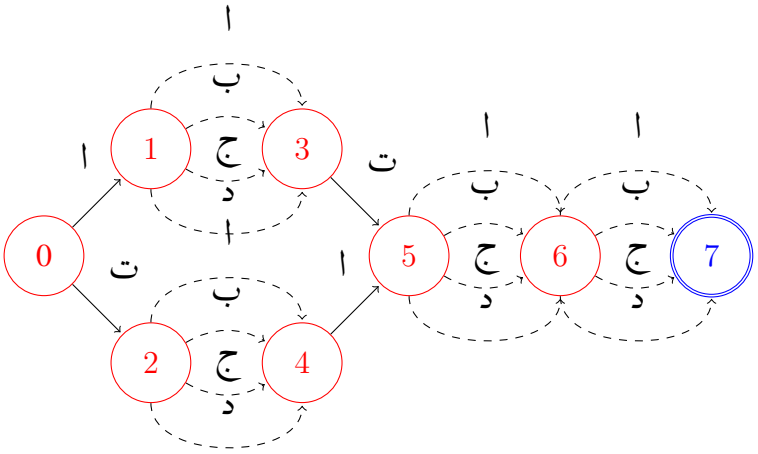



Figure 4.1: A DeFSA that accepts all the surface words derived from the patterns افتعل and تفاعل, the dotted edges between two states $q_i$ and $q_j$ represent the degeneracy case $p \xrightarrow{*} q$, where the $*$ represent any letter $\alpha$ such that $\alpha \in S$ and $S \subseteq \Sigma$

## 4.4 DeFSA: Degenerate Finite State Automaton

In this section, we present our proposed Degenerate Finite State Automaton, i.e., DeFSA. We first discuss how the DeFSA is constructed. Then we discuss how the matching process is done in DeFSA. Finally we discuss how we implement DeFSA using a DAWG.

### 4.4.1 DeFSA Construction

The DeFSA construction is identical to the standard FSA construction. However, the DeFSA is constructed from the template dictionary rather than from the word list directly. Briefly the process is as follows. At first, the solid and degenerate characters are identified from the template dictionary. Then an extended alphabet is constructed considering both the solid and non-solid characters identified above. Then we simply construct the DeFSA based on the template dictionary on the extended alphabet. An example of the whole process is given in a later section.

### 4.4.2 Matching Process

Once the DeFSA is constructed the matching process is done as follows. Suppose we want to validate a given word $w$. First we map $w$ from the original alphabet to the new extended alphabet, i.e., build the template of the word. Clearly, this can be easily done using the AMA. But we will shortly discuss how this can be done without the help of AMA. Suppose $w'$ is the transformed word. Then all we need is to check, whether $w'$ is accepted by the DeFSA just like a usual FSA.

### 4.4.3 DeFSA Construction Example

Figure 4.1 shows the diagram of DeFSA $A$ that accepts all the words (surface-forms) derived from the following two templates $t_1 = Alif + C_1 + Taa + C_2 + C_3$ (افتعل) and $t_2 = Taa + C_1 + Alif + C_2 + C_3$ (تفاعل). First we identify the solid characters and the degenerate characters in the templates. Here the solid characters are $Alif, Taa$. The characters $C_1, C_2$ and $C_3$ are degenerate characters and can be written as $*$ (don't care), which will take any character in the (Arabic) alphabet. So $t_1$ and $t_2$ can be rewritten as $t_1 = Alif + * + Taa + * + *$ and $t_2 = Taa + * + Alif + * + *$.

Next we create the new alphabet including the degenerate characters with the set of solid characters, i.e., $\{Alif, Taa\}$. Here, we have only one degenerate character, i.e., $*$. So, our extended character set is $\{Alif,\ Taa,\ *\}$. Now we can see that each of the two templates represents a path in $A$ (see Figure 4.1 and the transition table of $A$ in Table 4.1) from the start state to the accepting state. The transition table of $A$ is given in Table 4.1.

To understand how the matching process is done, recall that during the construction of $A$, we have used the following mapping rule: *convert every letter in the original alphabet to the newly introduced letter* $*$ *except for the two letters* $Alif$ *and* $Taa$. Let us call this mapping rule $R_1$. Now to check if a given word $w$ exists or not, we first apply $R_1$ to transform the the word $w$ from the original alphabet to the new alphabet $\Sigma = \{Alif,\ Taa,\ *\}$. This transformation can be done simply by iterating through each character $w[i]$ in the word $w$ and change the char $w[i]$ to $*$ if $w[i] \notin \{Alif,\ Taa\}$. The resulting word $w'$ can now be passed to $A$ in the same way as is done for any standard FSA.

| | Alif | Taa | * |
|---|---|---|---|
| $\rightarrow 0$ | 1 | 2 | 1 |
| 1 | 3 | 3 | 3 |
| 2 | 4 | 4 | 4 |
| 3 | 0 | 5 | 0 |
| 4 | 5 | 0 | 0 |
| 5 | 6 | 6 | 6 |
| 6 | 7 | 7 | 7 |
| 7 | 0 | 0 | 0 |

Table 4.1: $Q = \{1, 2, 3, 4, 5, 6, 7\}$, $\Sigma = \{Alif,\ Taa,\ *\}$, start state $= 0$,final state(s) $= \{7\}$, The $\rightarrow$ indicates the start state

### 4.4.4 DAWG Implementation of DeFSA

In this section, we describe the details of the DAWG implementation of DeFSA. The main motivation for implementing DeFSA using a DAWG is to achieve high speed and low memory footprint. So we make an effort to implement DAWG in the most efficient way. To this end we employ the so called bit-counting algorithm [Knu09] in our DAWG implementation. To implement the bit-counting algorithm efficiently, we use table lookups based on a pre-computed table as implemented in [EQ00]. Bit-counting method simply refers to counting the number of 1's in the binary representation of an integer. The concept is also known as the *Hamming weight* of a string, which is the number of symbols that differs from the zero-symbol of the alphabet used. It is thus equivalent to the Hamming distance from the all-zero string of the same length.

However, to be able to efficiently implement the bit-counting algorithm we need to be able to compactly represent the characters of the alphabet. While this is easy for the English alphabet, for the Arabic alphabet this is not straightforward as discussed below. While the English alphabet has 26 fixed letters, the Arabic alphabet consists of 42 different ligatures including 28 letters, 8 diacritics and multiple ligatures for some of the letters (e.g., there are five different types of the letter *Hamza*). Also, in the English charset page, the English alphabet (i.e., the characters A-Z) is represented in a nice compact and continuous range starting at 65 and ending at 90 (A=65, B=66, C=67 ...Z=90). But, the Arabic *charset windows-1256* table is not continuous. It starts at 193 and ends at 237 with a few entries in between either unused or allocated for symbols or chars from other languages. To alleviate these issues we first need to transform the representation of the Arabic alphabet to a more compact form.

### 4.4.5 Alphabet Compaction and Mapping

The alphabet compaction is in fact achieved automatically because we construct the DeFSA from the template dictionary rather than from the word list directly. Therefore, the alphabet used in the DAWG is the alphabet of the template dictionary. The following example illustrates how the alphabet gets compacted through this process.

**Example 4.4.1** *Consider the alphabet* $\{A, B, C, D, E\}$ *with the following word list* $\{AA, AB, AC, AD, BA, BB, BC, BD\}$. *Clearly, this word list is equivalent to the template list* $\{A*, B*\}$ *where* $*$ *is the degenerate character* $[ABCD]$. *The list* $\{A*, B*\}$ *represents the template dictionary of the original word list. Clearly, the alphabet of the template dictionary is* $\{A, B, *\}$*, which is smaller than the original alphabet of the word list: here the alphabet size is reduced from 5 to 3.*

Applying this technique and using the well-researched [AI12a, AB07, AJ87] template dictionary we can reduce the Arabic alphabet size from 42 down to 27. However, we still need a mapping to efficiently implement the bit-counting algorithm. The bit-counting algorithm requires that the range of the alphabet starts at zero. This is done by a mapping of the integer values of the characters. In what follows we will refer to this mapping as the *Zero based Alphabet Mapping*.

At this point, a brief discussion is in order. We have mentioned earlier that we need to map the word to be validated from the original alphabet to the new alphabet. While

this can be done using the AMA, the very idea of proposing and implementing DeFSA is to avoid the use of a heavy machinery like AMA for validating the words. So, there is no point proposing the DeFSA if we need to take the aid of AMA for validation. To this end, we can simply avoid taking the service of the AMA by implementing a mapping as mentioned above (i.e., Zero based Alphabet Mapping), and this is what we do in our implementation.

In fact we need to employ a two stage mapping here. At the first stage, we have a mapping to transform the word from the original Arabic alphabet to the alphabet of the template dictionary. This has to be followed by the Zero based Alphabet Mapping. Both stages can be done in one go at construction time. During the matching process we already have the (2-stage) mapping stored so we just need to encode the given word according to the mapping.

| Experiment Results | | | | | |
|---|---|---|---|---|---|
| (1) - Comparison between regular FSA and DeFSA size | | | | | |
| verb category | text (KB) | keyword list (KB) | FSA (KB) | DeFSA (KB) | no. of words |
| 1 | 349,967 | 76,900 | 305,260 | 78,899 | 5,144,386 |
| 2 | 305,025 | 74,808 | 298,553 | 91,528 | 5,007,925 |
| 3 | 284,479 | 73,842 | 249,101 | 57,968 | 4,938,904 |
| 4 | 325,816 | 75,293 | 252,712 | 77,160 | 5,039,476 |
| 5 | 446,457 | 79,898 | 321,273 | 76,924 | 5,330,953 |
| (2) - Comparison between DeFSA and D-DeFSA size | | | | | |
| verb category | DeFSA (KB) | D-DeFSA (KB) | | | |
| 1 | 78,899 | 298 | | | |
| 2 | 91,528 | 359 | | | |
| 3 | 57,968 | 205 | | | |
| 4 | 77,160 | 302 | | | |
| 5 | 76,924 | 281 | | | |

Table 4.2: Comparison between FSA, DeFSA and D-DeFSA

## 4.5 Experiment

We have done experiments to investigate the performance of DeFSA and its DAWG implementation (D−DeFSA). We have compared them with a normal FSA which is constructed with a direct implementation of the morphological rules. The experiments are carried out on a Windows Server 2008 R2 64-bit Operating System, with Intel(R) Core(TM) i7 2600 processor @ 3.40GHz having an installed memory (RAM) of 8.00 GB. The FSA, DeFSA and the D−DeFSA have been implemented in $C\#$ using Visual Studio 2010. The experiments are carried out on the Arabic verb categories 1-5. The experiments are done by selecting randomly 100 files (word lists generated from 100 roots) from each verb category followed by the construction of the DeFSA for each set of files. Subsequently DeFSA was converted to D−DeFSA by implementing it as a DAWG. The results are presented in Table 4.2.

As can be seen in Table 4.2, DeFSA outperforms the alternative FSA implementation and the memory footprint reduction achieved by D−DeFSA is extraordinary.

DeFSA, and its DAWG implementation D−DeFSA, have succeeded in compactly represent sets of strings that share common aligned parts yet differ in affixes (including infixes), by employing the concept of degenerate symbols, the remaining characters are replaced by degenerate symbols allowed at certain positions. In this way, a smaller set of degenerate strings is obtained, leading to savings in the representation space. Taking advantage of this structured degeneracy phenomenon, which is particularly relevant in Arabic, we have been able to counter the size and speed issues of the existing and more general purpose Arabic Morphological Analyzer, "AMA", while analyzing and validating Arabic words.

# Chapter V

# Concluding Remarks

**- Article 1: Inferring an Indeterminate String from a Prefix Graph.**

This article provides the first algorithm to reverse engineer a data structure to a string in its full generality; we hope that future work will show how to generalize data structures currently restricted only to regular strings, and to solve similar problems that arise. In particular, for the reverse engineering of the prefix array, we wonder if a more efficient algorithm can be found than the one described here, perhaps one that reduces to $\mathcal{O}(n)$ for regular strings.

**- Article 2: Computing Covers Using Prefix Tables:**

There are several data structures related to the cover array whose computation may now be contemplated in the context of indeterminate strings. For example, a recent paper [FIK+13] introduces new forms of "enhanced" cover array that are efficiently computed using the border array; using the cover array instead would open the way for computation of variants of these structures also for indeterminate strings. Similarly, another recent paper [CCI+11] proposes efficient algorithms for the computation of "seed" arrays (a ***seed*** of a string $\boldsymbol{x}$ is a cover of some superstring of $\boldsymbol{x}$) — these algorithms also may be similarly extended.

**- Article 3: Algorithms for Longest Common Abelian Factors:**

In this article, we present a simple quadratic running time algorithm for the `LCAF` problem for binary strings and also a sub-quadratic running time algorithm by using the index data structure of Moosa and Rahman [MR10]. Both solution have linear space requirement.

A variant of the above sub-quadratic algorithm can be obtained replacing the Moosa and Rahman [MR10] index with the approximate indexing data structure of Cicalese et al. in [CLWY12] in order to compute the $minOne(A, \ell)$, $maxOne(A, \ell)$, $minOne(B, \ell)$, $maxOne(B, \ell)$ values. Even if such computation, for a fixed $\ell$, runs in near linear time, the presence of false positive values leads to a quadratic total time algorithm in the worst case due to a linear checking step for any $\ell$ value. It is not trivial to obtain a better time bound for this algorithm variant. Furthermore, we present a variant of the quadratic solution that is experimentally shown to achieve a better time complexity of $\mathcal{O}(\sigma\ n\ \log n)$, where $\sigma$ is the alphabet of cardinality, i.e., $\mathcal{O}(n\ \log n)$ for a constant alphabet.

An interesting idea is that introducing the idea of ***heaviest*** longest common abelian factor: a factor $u$ is heavier than $v$ if $|u| = |v|$ and $one(u) > one(v)$. So, suppose we have two common abelian factors of length LCAF between two strings. Suppose these two factors are 0011100 and 1001110. Then both are longest common abelian factors but the latter one is a ***heaviest*** longest common abelian factor.

**- Article 4: Maximal Palindromic Factorization:**

We answer a recent question raised during StringMasters, Verona, Italy - 2013: does there exist an algorithm to compute the maximal palindromic factorization of a finite string? Namely, given a finite string, find the smallest set (minimum number of palindromic factors), such that the string is covered by that set of factors with no overlaps. We answer the previous question affirmatively by providing a linear-time algorithm that computes the *maximal palindromic factorization (MPF)* of a string (the algorithms is evaluated with respect to the length of the given string). We wonder if MPF algorithm can be extended to find ***maximal distinct palindromic factorization set***. We will focus on this problem in a future work.

Furthermore, we showed that MPF algorithm can be extended to biological palindromes, where the word reversal is defined in conjunction with the complementarity of nucleotide letters: $c \leftrightarrow g$ and $a \leftrightarrow t$ (or $a \leftrightarrow u$, in the case of RNA).

In [AIR13], we introduced in the concept of palindromic cover of string and how can it be computed and modeled using graphs.

Recently, [ISI+14] have extended our results in [AIR13] and presented an online algorithm for computing the smallest palindromic factorizations in $\mathcal{O}(n \log n)$ and answered successfully to our question about the palindromic covers with an algorithm for computing the smallest palindromic covers in linear time and space.

**- Article 5: Lyndon Fountains and the Burrows-Wheeler Transform:**

We have introduced a novel concept: the Lyndon fountain related to the factorizations of all cyclic rotations of a given Lyndon word. These factorizations yield

many further Lyndon words; we have analyzed combinatorial properties with respect to these fountains and outlined related factorization and BWT algorithms. We propose that applications of Lyndon fountains may arise in bioinformatics in relation to the Burrows-Wheeler Transform.

**- Article 6: Specialized Border and Suffix Arrays:**

In this article, we have extended two well-known data structures in stringology. We first have adapted the concept of a border array to introduce the ***Lyndon Border Array*** $\mathcal{L}\beta$ of a string $\boldsymbol{s}$, and have described a linear-time and linear-space algorithm for computing $\mathcal{L}\beta(\boldsymbol{s})$. Furthermore, we have defined the ***Lyndon Suffix Array***, which is an adaptation of the classic suffix array. By modifying the linear-time construction of Ko and Aluru [KA03] we similarly achieve a linear construction for our Lyndon variant. We also present a simpler algorithm to construct a ***Lyndon Suffix Array*** from a given Suffix Array.

The potential value of the Lyndon Border Array is that it allows for deeper burrowing into a string to yield paired Lyndon patterned substrings. The Lyndon suffix array lends itself naturally to searching for Lyndon patterns in a string. If the given text or string has a sparse number of Lyndon words (as likely in English literature due to the vowels $a, e$ often occurring internally in words), then the Lyndon suffix array may offer efficiencies. Polyrhythms, or cross-rhythms, are when two or more independent rhythms play at the same time – nested Lyndon suffixes can exist in these rhythms. We propose that applications of these specialized data structures might arise in the context of the relationship existing between de Bruijn sequences and Lyndon words [FM78].

**- Article 7: Simple Linear Comparison of Strings in *V*-order:**

Lexicographic orderings have also been considered in the case of parallel computations: for instance, an optimal algorithm for lexordering $n$ integers is given in [Ili86], and parallel Lyndon factorization in [DIS94, DDIS13]. Analogously, we propose future research into parallel forms of ***V***-order-ordering strings.

**- Article 1: SimpLiSMS: Structured Motifs Searching:**

In this article, we have presented ***SimpLiSMS***, a simple and lightweight algorithm for structured motif searching that runs extremely fast in practice. We

have also implemented ***SimpLiSMS-P***, a parallel version of ***SimpLiSMS*** that runs even faster. We augmented our algorithm with the capability to enable search for degenerate motifs. Our immediate target is to implement a software capable to search directly from databases like PROSITE.

**- Article 2: On the Repetitive Collection Indexing Problem:**

In this article, we have presented a differential compression method that is based on the and locations and types of differences between each sequence in a genomic similar collection and its reference sequence is presented. This method is simple, universal in that it does not depend on the statistics of the data set and could achieve higher compression. Our algorithm differs in that we focus not only on compression ratio only but also on:

1. Compression and decompression speed or the memory use during the compression process; while other algorithms focus on compression ratio only.
2. Detecting inter-genome redundancy efficiently, while in previous work, most effort has been put to represent a single genome considering the compression ratio.
3. In our scheme, absolute position values can be reached without traversing all previous variations, which is required of previous methods.
4. Achieving good compression ratios while efficient extraction of individual sequences is possible.
5. Allowing users to quickly compute statistics on various types of variations and to retrieve complete or partial genomic sequences.

**- Article 1: Arabic Morphology Analysis and Generation:**

AMA has many functions. It can validate a word (accept or reject) and also can generate different forms of a given word. It also can be used as a stemmer (this functionality is discussed, in detail, in the next article): give it a word as input and ***AMA*** will return the stem of that word. Also ***AMA*** can return all the word annotations such as morphological properties (category, type, part of speech tags) of the input words. Another way to use ***AMA*** is to give it the annotations as input and AMA will return all the words that match these annotations.

Further more. ***AM*** can be used for measuring lexical semantic relatedness, we will investigate this aspect, in more details, in future work.

**- Article 2: Novel Arabic Language Stemmer:**

We have presented a new stemming technique by augmenting the Arabic Morphology Analyzer ***AMA*** [AI12a] with an extra stemming model, we build an expert system from ***roots + patterns + rules*** dictionaries derived from morphological and phonological representation of the language in order to minimise both stemming errors and stemming cost.

Further more, we would like to extend our stemmer to include rules for broken plurals, collective nouns, Nisba, cardinal/ordinal numerals and all other conjugations of irregular verbs (doubled, Weak, Assimilated, Hollow and Defective) which don't have standard patterns.

The experiment results, show that in comparison with the other stemmers, the performance superiority of the proposed ***AMA*** stemmer is clearly evident.

**- Article 3: Improved noisy channel model for Arabic spelling correction:**

In this article, we proposed an original way to automatically learn the costs of the edit operations, by taking into account the number of edges separating the words in to lexicon and affixation segments. Plugging in our weighted edit distance, we show that these edit costs allow us to improve the accuracy of the noisy channel model for Arabic spell checking. A complete and conclusive word list of Arabic is almost impossible to obtain due to the high morphological productivity of the language and the many constraints which even experienced linguists might not be able to explicitly formulate.

Furthermore, we introduce a practical component for automatically detecting and correcting single candidate errors using Aho-Corasick automaton to store errors dictionaries, of pairs of correct words and their erroneous matches, in a more compact manner. The input text can then be threaded against the Aho-Corasick automaton and verified against the valid list of words.

In this article, we proposed an expert system for spelling correction using noisy channel model for Arabic language, by providing an algorithm to find the optimal costs of a set of string edit rules based on the characteristics of Arabic

language. We follow [RY98] in using dynamic programming to find the permutation of edit operations with the minimum total edit cost for a pair of strings. The system is based on deep syntactic and semantic analysis and relies on a feature relaxation approach capable of detecting and suggesting improvements of spelling mistakes in Arabic words that goes undetected using conventional methods. By studying the results obtained using the improved model, it has been shown to be superior to the widely well known ***Hunspell*** spell checker [Hun13] and outperformed the baseline single edit distance.

We believe this to be the first spelling correction system designed, specifically for Arabic, for detecting and correcting such errors that usually go undetected using existing spell checkers.

Borrowing words from other languages is an important source of new words. An immediate goal for further work is to build similar system for correcting foreign words borrowed from other languages.

Lastly, an interesting potential direction of spelling correction is personalization: to tune the error model to mistakes often made by that person and adapting the language model to the author or the topic of discourse or by people of certain cultural backgrounds.

**- Article 4: Degenerate Finite State Automata for Arabic Morphology:**

In this article, we have proposed Degenerate Finite State Automaton (`DeFSA`), to facilitate the expression of various non-concatenative morphological phenomena in an efficient way. Our conclusion is that `DeFSA` implementation remains superior to its alternative, FSA, with respect to the true representation of linguistic knowledge, and is therefore more efficient for large-scale grammars. The implementation of `DeFSA` with DAWG, i.e., `D-DeFSA`, turns out to be even better in terms of speed and memory.

Finally, we note here that, our approach is not only limited to the Arabic Language; rather it is supposed to work for any language that follows specific morphological rules in the word generation process and can be represented as a set of morphological patterns or regular expressions.

# References


[AAA+92] Eneko Agirre, Iñaki Alegria, Xabier Arregi, Xabier Artola, Arantza Díaz de Ilarraza Sánchez, Montse Maritxalar, Kepa Sarasola, and Miriam Urkia. Xuxen: A spelling checker/corrector for basque based on two–level morphology. In *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 119–125, 1992. 206

[AB07] Marwan Al-Bawab. In *Arabic Morphology System*, volume vol.1. 2007. 188, 190, 231

[AB12] Houda Amraoui and Riadh Bouslimi. Fault detection system for arabic language. *Computing Research Repository*, abs/1203.2498, 2012. 210

[ABM08] Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows–Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Publishing Company, 2008. 106, 133

[Abr87] K. Abrahamson. Generalized string matching. *SIAM Journal of Computing*, 16(6):1039–1051, 1987. 48, 65

[AC75] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975. 22, 23, 158, 217

[AC95] Alberto Apostolico and Maxime Crochemore. Fast parallel Lyndon factorization with applications. *Mathematical Systems Theory*, 28(2):89–108, 1995. 26, 117

[ADI+09] P. Antoniou, J. W. Daykin, C. S. Iliopoulos, D. Kourie, L. Mouchard, and S. P. Pissis. Mapping uniquely occuring short sequences derived

from high throughput technologies to a reference genome. In *Proceedings of the 9th IEEE International Conference on Information Technology and Applications in Biomedicine (ITAB 2009)*, 2009. DOI: 10.1109/ITAB.2009.5394394. 106

[AE90] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasi–periodicities in strings. Technical Report 90.5, The Leonadro Fibonacci Institute, Trento, Italy, 1990. 64

[AF02] Mohammed Aljlayl and Ophir Frieder. On arabic search: improving the retrieval effectiveness via a light stemming approach. In *In Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA*, pages 340–347, 2002. 194, 195

[AFI91] A. Apostolico, M. Farach, and C.S. Iliopoulos. Optimal superprimitivity testing for strings. *Inform. Process. Lett.*, 39:17–20, 1991. 64

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, 1974. 64

[AI12a] Ali Alatabbi and Costas S. Iliopoulos. Morphological analysis and generation of arabic language. *International Journal of Information Technology & Computer Science (IJITCS)*, 3, 2012. 193, 199, 206, 207, 220, 222, 231, 239

[AI12b] Ali Alatabbi and Costas S. Iliopoulos. Novel arabic language stemmer. In *Proceedings of the International Conference on Computer Science, Engineering & Technology (ICCSET)*, 2012. 199, 206

[AI14] Ali Alatabbi and Costas S. Iliopoulos. Statistical paradigm of arabic language (in review). 2014. 209

[AIR13] Ali Alatabbi, Costas S. Iliopoulos, and M. Sohel Rahman. Maximal palindromic factorization. In *Proceedings of (PSC) The Prague Stringology Conference*, 2013. 236

[AIW11] Heba Afify, Muhammad Islam, and Manal Abdel Wahed. DNA lossless differential compression algorithm based on similarity of genomic sequence database. *International Journal of Computer Science and Information Technology (IJCSIT)*, 3(4), 2011. 171

[AJ87] Abd Al-Qahir Al-Jurjani. *The key for Arabic morphology*. Al–Risala Publishing Association, 1987. Edited by Ali Al–Hamed. 188, 190, 231

[AJ88] Andrew W. Appel and Guy J. Jacobson. The world's fastest scrabble program. *Programming Techniques and Data Structures*, 31(5), 1988. 19, 226

[ALV90] Amihood Amir, Gad M. Landau, and Usi Vishkin. Efficient pattern matching with scaling. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 344–357. Society for Industrial and Applied Mathematics, 1990. 22

[ARS15] Ali Alatabbi, M. Sohel Rahman, and William F. Smyth. Inferring an indeterminate string from a prefix graph. *J. Discrete Algorithms*, 32(0):6–13, 2015. StringMasters 2012 & 2013 Special Issue (Volume 2). 66

[ASL08] Eiman Tamah Al-Shammari and Jessica Lin. Towards an error–free arabic stemming. *CIKM–iNEWS*, pages 9–16, 2008. 196

[Bay63] T. Bayes. An essay towards solving a problem in the doctrine of chances. *Phil. Trans. of the Royal Soc. of London*, 53:370–418, 1763. 211

[BBB+09] Timothy L. Bailey, Mikael Bodén, Fabian A. Buske, Martin C. Frith, Charles E. Grant, Luca Clementi, Jingyuan Ren, Wilfred W. Li, and William Stafford Noble. MEME SUITE: tools for motif discovery and searching. *Nucleic Acids Research*, 37(Web-Server-Issue):202–208, 2009. 149

[BCFL10] Peter Burcsi, Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. On table arrangements, scrabble freaks, and jumbled pattern matching. volume 6099 of *Lecture Notes in Computer Science*, pages 89–101. Springer, 2010. 84

[BFC00] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. 130

[BG95] Dany Breslauer and Zvi Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14:355–366, October 1995. 95

[BGM11] Dany Breslauer, Roberto Grossi, and Filippo Mignosi. Simple real–time constant–space string matching. In *Symp. on Combinatorial Pattern Matching*, pages 173–183, 2011. 26, 95, 106, 134

[BGVW12] Philip Bille, Inge Li Gortz, Hjalte Wedel Vildhoj, and David Kofoed Wind. String matching with variable length gaps. *Theor. Comput. Sci.*, 443:25–34, 2012. 149, 150

[BIST03] Hideo Bannai, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Inferring strings from graphs and arrays. In Branislav Rovan and Peter Vojt's, editors, *Symp. on Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003. 50

[BJ06] L. Bahl and F. Jelinek. Decoding for channels with insertions, deletions, and substitutions with applications to speech recognition. *IEEE Trans. Inf. Theor.*, 21(4):404–411, September 2006. 216

[BK03] Kenneth R. Beesley and Lauri Karttunen. *Finite State Morphology*. Center for the Study of Language and Information, April 2003. 183

[BKS13] Widmer Bland, Gregory Kucherov, and W. F. Smyth. Prefix table construction and conversion. *Proc. 24th Internat. Workshop on Combinatorial Algs. (IWOCA)*, pages 41–53, 2013. 49, 65

[BLPR09] Srecko Brlek, Jacques-Olivier Lachaud, Xavier Provencal, and Christophe Reutenauer. Lyndon + Christoffel = digitally convex. *Pattern Recognition*, 42(10):2239–2246, 2009. 134

[BM77] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977. 22, 154, 158

[BM00] Eric Brill and Robert C. Moore. An improved error model for noisy channel spelling correction. ACL, 2000. 205, 215, 219

[BM02] Mikhail Bilenko and Raymond J. Mooney. Learning to combine trained distance metrics for duplicate detection in databases. Technical report, 2002. 205, 216

[Bre92] D. Breslauer. An on–line string superprimitivity test. *Inform. Process. Lett.*, 44(6):345–347, 1992. 64

[Bre06] C. Brezina. *Al–Khwarizmi: The Inventor of Algebra*. Great Muslim philosophers and scientists of the Middle Ages. Rosen Publishing Group, 2006. 16

[BRS09] Md. Faizul Bari, Mohammad Sohel Rahman, and Rifat Shahriyar. Finding all covers of an indeterminate string in $o(n)$ time on average. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2009*, pages 263–271, 2009. 65, 66, 71, 74

[BS08] Francine Blanchet-Sadri. *Algorithmic Combinatorics on Partial Words*. Chapman & Hall CRC, 2008. 48, 65

[BSBW14] Francine Blanchet-Sadri, Michelle Bodnar, and Benjamin De Winkle. New bounds and extended relations between prefix arrays, border arrays, undirected graphs, and indeterminate strings. In N. Portier and E. Mayr, editors, *Proc. 31st Symp. on Theoretical Aspects of Computer Science*, pages 162–173, 2014. 50, 60, 62

[BT10] Philip Bille and Mikkel Thorup. Regular expression matching with multi–strings and intervals. In Moses Charikar, editor, *ACM-SIAM Symp. Discrete Algs.*, pages 1297–1308. SIAMPublications, 2010. 150

[BW94] M. Burrows and D. Wheeler. A block–sorting lossless data compression algorithm. Technical report, Digital SRC Research Report 124, 1994. 19, 23, 95, 106, 107

[BWML06] Timothy L. Bailey, Nadya Williams, Chris Misleh, and Wilfred W. Li. MEME: discovering and analyzing DNA and protein sequence motifs. *Nucleic Acids Research*, 34(Web-Server-Issue):369–373, 2006. 149

[BY89] Ricardo A. Baeza-Yates. Algorithms for string searching: A survey. *SIGIR Forum*, 23(3–4):34–58, 1989. 22

[BYG96] Ricardo A. Baeza-Yates and Gaston H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. *J. ACM*, 43(6):915–936, 1996. 171

[CCI+11] Michalis Christou, Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Bartosz Szreder, and Tomasz Walen. Efficient seeds computation revisited. *Proc. 22nd CPM, Raffaele Giancarlo and Giovanni Manzini (eds.), Lecture Notes in Computer Science, LNCS 6661, Springer-Verlag, 350–363*, 2011. 235

[CCR09] Julien Clément, Maxime Crochemore, and Giuseppina Rindone. Reverse engineering prefix tables. In *Proc. 26th Symp. on Theoretical Aspects of Computer Science*, pages 289–300, 2009. 50, 62

[CDP05] M. Crochemore, J. Désarménien, and D. Perrin. A note on the Burrows–Wheeler Transformation. *Computing Research Repository*, abs cs 0502073, 2005. 95, 106, 133

[CFL58] K. T. Chen, R. H. Fox, and R. C. Lyndon. Free differential calculus, iv – the quotient groups of the lower central series. *Ann. Math.*, pages 68:81–95, 1958. 106, 117, 119, 134

[CFL09] Ferdinando Cicalese, Gabriele Fici, and Zsuzsanna Lipták. Searching for jumbled patterns in strings. In Jan Holub andJan Zd árek, editor, *Stringology*, pages 105–117. Prague Stringology Club, Department of Computer Scienceand Engineering, Faculty of Electrical Engineering, CzechTechnical University in Prague, 2009. 84

[CFMPN10] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed $q$–gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*, pages 86–91, 2010. 24, 172

[CG91] Kenneth W. Church and William A. Gale. Probability scoring for spelling correction. *Statistics and Computing*, 1:93–103, 1991. 205, 215

[CHC10] Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao. Identifying approximate palindromes in run–length encoded strings. In *Proceedings of 21st International Symposium, ISAAC 2010, Jeju, Korea, December 15–17, 2010*, pages 339 – 350, 2010. 95

[Che04] M. Chemillier. Periodic musical sequences and Lyndon words. *Soft Comput.*, 8(9):611–616, 2004. 26, 106, 117, 134

[CHIR14] Shihabur Rahman Chowdhury, Md. Mahbubul Hasan, Sumaiya Iqbal, and M. Sohel Rahman. Computing a longest common palindromic subsequence. *Fundamenta Informaticae*, page To Appear, 2014. 95

[CHL07] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007. 23, 36, 38, 118

[CHO05a] S.K. Chan, B. He, and I. Ounis. An in–depth survey on the automatic detection and correction of spelling mistakes. In *Proceedings of the 5th Dutch–Belgian Information Retrieval Workshop (DIR'05)*, Utrecht, Netherlands, 2005. 214

[Cho05b] Charles Choi. DNA palindromes found in cancer. *Genome Biology*, 6, 2005. 96

[CIS08] Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the lempel ziv factorization. pages 482–488. IEEE Computer Society, 2008. 95

[CL04] Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004. 21

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001. 83

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. 44

[CLWY12] Ferdinando Cicalese, Eduardo Sany Laber, Oren Weimann, and Raphael Yuster. Near linear time construction of an approximate index forall maximum consecutivesub–sums of a sequence. In Juha Kärkkäinen and Jens Stoye, editors, *Symp. on Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 149–158. Springer, 2012. 235

[CP91] Maxime Crochemore and Dominique Perrin. Two–way string matching. *J. ACM*, 38(3):651–675, 1991. 26, 95, 106, 134

[CR94] Maxime Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994. 22

[CR02] Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002. 117

[CRSW14] Manolis Christodoulakis, P. J. Ryan, W. F. Smyth, and Shu Wang. Indeterminate strings, prefix arrays and undirected graphs. *CoRR abs/1406.3289*, 2014. 47, 49, 50, 51, 52, 53, 56, 62, 65, 66

[CS04] Maxime Crochemore and Marie-France Sagot. 1. motifs in sequences. *Compact Handbook of Computational Biology*, page 47, 2004. 148

[CT03] Marc Chemillier and Charlotte Truchet. Computation of words satisfying the "rhythmic oddity property" (after simha arom's works). *Inform. Process. Lett.*, 86(5):255–261, 2003. 135

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, March 1964. 22, 37, 219

[Dav62] Leon Davidson. Retrieval of misspelled names in an airlines passenger record system. *Commun. ACM*, 5(3):169–171, March 1962. 205

[Day85] D. E. Daykin. Graphs and order, ordered ranked posets, representations of integers and inequalities from extremal poset problems. *Proceedings of a Conference in Banff, Canada (1984), NATO Advanced Sciences Institutes Series C: Mathematical and Physical Sciences 147 ( I. Rival (ed.); Reidel, Dordrecht–Boston, 1985)*, pages 395–412, 1985. 134

[Day11] D. E. Daykin. Algorithms for the Lyndon unique maximal factorization. *J. Combin. Math. Combin. Comput.*, 77(3):65–74, 2011. 134

[DD96] T.-N. Danh and D.E. Daykin. The structure of $V$–order for integer vectors. *Congr. Numer.Ed. A.J.W. Hilton. Utilas Mat. Pub. Inc., Winnipeg, Canada, 113 (1996)*, pages 43–53, 1996. 134, 135, 137

[DD97] T.-N. Danh and D. E. Daykin. Ordering integer vectors for coordinate deletions. *J. London Math. Soc.*, 55(2):417–426, 1997. 134

[DD03] D.E. Daykin and J.W. Daykin. Lyndon–like and $V$–order factorizations of strings. *J. Discrete Algorithms*, (1):357–365, 2003. 135, 136, 137

[DDIS13] D. E. Daykin, J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Generic algorithms for factoring strings. *Proc. Memorial Symp. for Rudolf Ahlswede, H. Aydinian, F. Cicalese & C. Deppe, (eds.), Lecture Notes in Comput. Sci. Festschrifts Information Theory, Combinatorics, and Search Theory: In Memory of Rudolf Ahlswede*, 7777:402–418, 2013. 136, 237

[DDS11] David E. Daykin, Jacqueline W. Daykin, and William F. Smyth. String comparison and Lyndon–like factorization using $V$–order in linear time. In *Symp. on Combinatorial Pattern Matching*, volume 6661, pages 65–76, 2011. 135, 136, 137, 140

[DDS13] D. E. Daykin, J. W. Daykin, and W. F. Smyth. A linear partitioning algorithm for hybrid Lyndons using $V$–order. *Theoret. Comput. Sci.*, 483:149–161, 2013. 134, 135, 136, 137

[DEDS09] J. W. Daykin D. E. Daykin and W. F. Smyth. Combinatorics of unique maximal factorization families (umffs). *Fund. Inform. 97–3, Special Issue on Stringology, R. Janicki, S. J. Puglisi and M. S. Rahman (eds.)*, pages 295–309, 2009. 134

[DHK94] Rickard Domeij, Joachim Hollman, and Viggo Kann. Detection of spelling errors in swedish not using a word list en clair. *J. Quantitative Linguistics*, 1:1–195, 1994. 205

[DIS94] J. W. Daykin, C. S. Iliopoulos, and W. F. Smyth. Parallel ram algorithms for factorizing words. *Theoret. Comput. Sci.*, 127(1):53–67, 1994. 134, 237

[DLL05] Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005. 50

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000. 226

[DO07] Kareem Darwish and Douglas W. Oard. 13 adapting morphology for arabic information retrieval. *Arabic Computational Morphology*, 38, 2007. 196

[DP99] X. Droubay and G. Pirillo. Palindromes and Sturmianwords. *Theoretical Computer Science*, 223:73–85, 1999. 95

[DR04] O. Delgrange and E. Rivals. Star: an algorithm to search for tandem approximate repeats. *Bioinformatics*, 20(16):2812–2820, 2004. 26, 106, 117

[Dro95] Xavier Droubay. Palindromes in the fibonacci word. *Inform. Process. Lett.*, 55(4):217–221, 1995. 95

[DS14] J. W. Daykin and W. F. Smyth. A bijective variant of the Burrows–Wheeler transform using $V$–order. *Theoret. Comput. Sci.*, 531:7789, 2014. 128, 136

[Duv83] Jean-Pierre Duval. Factorizing words over an ordered alphabet. *J. Algorithms*, 4(4):363–381, 1983. 106, 115, 117, 119, 122, 134

[Ell85] R. S. Ellis. *Entropy, Large Deviations, and Statistical Mechanics*. Springer, 1985. 88

[EQ00] Eyas El-Qawasmeh. Performance investigation of bit–counting algorithms with a speedup to lookup table. *Journal of Research and Practice in Information Technology*, 32(3/44):215–230, 2000. 230

[Erd61] P. Erdös. Some unsolved problems. *Magyar Tud. Akad. Mat. Kutato. Int. Kozl.*, 6:221–254, 1961. 78

[FF03] William B. Frakes and Christopher J. Fox. Strength and similarity of affix removal stemming algorithms. *SIGIR Forum*, 37(1):26–30, 2003. 194, 200

[FGL+00] Frantisek Franek, Shudi Gao, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear time. *J. Comb. Math. Comb. Comput*, 42:223–236, 2000. 50

[FHIP10] T. Flouri, J. Holub, C. Iliopoulos, and S. Pissis. An algorithm for mapping short reads to a dynamically changing genomic sequence. In *Bioinformatics and Biomedicine (BIBM), 2010 IEEE International Conference on*, pages 133 –136, dec. 2010. 178

[FIK+13] Tomas Flouri, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Simon J. Puglisi, W.F. Smyth, and Wojciech Tyczynski. Enhanced string covering. *Theoretical Computer Science*, 506:102 – 114, 2013. 235

[FL10] Simone Faro and Thierry Lecroq. The exact string matching problem: a comprehensive experimental evaluation. *Computing Research Repository*, abs/1012.2547, 2010. 21

[FLR+99] Frantisek Franek, Weilin Lu, P. J. Ryan, W. F. Smyth, Yu Sun, and Lu Yang. Verifying a border array in linear timee (preliminary version). *Proc. 10th Australasian Workshop on Combinatorial Algs. (AWOCA) School of Computing, Curtin University of Technology*, pages 26–33, 1999. 50

[FM78] H. Fredricksen and J. Maiorana. Necklaces of beads in *k*colors and *k*–ary de bruijn sequences. *Discrete Mathematics*, 23(3):207–210, 1978. 26, 117, 237

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *IEEE Symp. on Foundations of Computer Science*, pages 390–398. IEEE Computer Society, 2000. 171

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. 19

[FNU02] Kimmo Fredriksson, Gonzalo Navarro, and Esko Ukkonen. Optimal exact and fast approximate two dimensional patternmatching allowing rotations. In Alberto Apostolico and Masayuki Takeda, editors, *Symp. on Combinatorial Pattern Matching*, volume 2373 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 2002. 22

[FP74] M.J. Fischer and M.S. Paterson. String matching and other products. In R.M. Karp, editor, *Complexity of Computation,*, pages 113–125. American Mathematical Society, 1974. 22, 48, 65

[FS06] Frantisek Franek and William F. Smyth. Reconstructing a suffix array. *Internat. J. Foundations of Computer Science*, 17(6):1281–1296, 2006. 50

[FU98] Kimmo Fredriksson and Esko Ukkonen. A rotation invariant filter for two-dimensional string matching. In Martin Farach-Colton, editor, *Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, pages 118–125. Springer Berlin Heidelberg, 1998. 22

[FW74] M.J. Fischer and R. Wagner. The string–to–string correction problem. *J. Assoc. Comput. Mach.*, 21:168–178, 1974. 22, 205

[Gal76] Zvi Galil. Real–time algorithms for string–matching and palindrome recognition. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 161–173. ACM, 1976. 95

[GBEB97] William Noble Grundy, Timothy L. Bailey, Charles Elkan, and Michael E. Baker. Meta-meme: motif-based hidden Markov models of protein families. *Computer Applications in the Biosciences*, 13(4):397–406, 1997. 149

[GBL95] M. Graves, E.R. Bergeman, and C.B. Lawrence. Graph database systems. *Engineering in Medicine and Biology Magazine, IEEE*, 14(6):737–745, 1995. 174

[GD11] Szymon Grabowski and Sebastian Deorowicz. Engineering relative compression of genomes. *Computing Research Repository*, abs/1103.2351, 2011. 172

[Gle06] Amy Glen. Occurrences of palindromes in characteristic sturmian words. *Theor. Comput. Sci.*, 352(1–3):31–46, 2006. 95

[Gra53] F. Gray. Pulse code communication. U.S. Patent 2 632 058, 1953. 134

[GS12] J. Y. Gil and D. A. Scott. A bijective string sorting transform. *Computing Research Repository*, abs/1201.3077, 2012. 26, 95, 106, 117

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Cambridge University Press, 1997. 19, 20, 23, 37, 41, 42, 78, 95, 97, 104

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 26(2):147–160, 1950. 22

[HBB+06] Nicolas Hulo, Amos Bairoch, Virginie Bulliard, Lorenzo Cerutti, Edouard De Castro, Petra S Langendijk-Genevaux, Marco Pagni, and

Christian JA Sigrist. The PROSITE database. *Nucleic acids research*, 34(suppl 1):D227–D230, 2006. 148, 149, 156

[HC03] Jan Holub and Maxime Crochemore. On the implementation of compact DAWG's. In Jean-Marc Champarnaud and Denis Maurel, editors, *Implementation and Application of Automata*, volume 2608 of *Lecture Notes in Computer Science*, pages 289–294. Springer Berlin Heidelberg, 2003. 227

[HCC09] Ping-Hui Hsu, Kuan-Yu Chen, and Kun-Mao Chao. Finding all approximate gapped palindromes. In *Proceedings of 20th International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16–18, 2009.*, pages 1084 – 1093, 2009. 95

[HHXZ10] Lenwood S. Heath, Ao Ping Hou, Huadong Xia, and Liqing Zhang. A compression algorithm supporting manipulation. In *Proc LSS Comput Syst Bioinform Conf*, pages 38–49, 2010. 171

[Hir77] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977. 22

[HS03] Jan Holub and W. F. Smyth. Algorithms on indeterminate strings. *Proc. 14th Australasian Workshop on Combinatorial Algs. (AWOCA)*, pages 36–45, 2003. 48, 65, 66

[HS08] Mihail Halachev and Nematollaah Shiri. Fast structured motif search in DNA sequences. In *Bioinformatics Research and Development, Second International Conference, BIRD 2008, Vienna, Austria, July 7-9, 2008, Proceedings*, pages 58–73, 2008. 150

[HSW08] J. Holub, W. F. Smyth, and S. Wang. Fast pattern–matching on indeterminate strings. *J. Discrete Algorithms*, pages 37–50, 2008. 65

[HU79] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison–Wesley Publishing Company, Reading, Massachusets, USA, 1979. 38

[Hun13] Hunspell. Hunspell spell checker 1.3.2, 2013. 222, 240

[HY07] Bassam Haddad and Mustafa Yaseen. Detection and correction of non–words in arabic: a hybridapproach. *Int. J. Comput. Proc. Oriental Lang.*, 20(4):237–257, 2007. 210

[Ili86] C.S. Iliopoulos. Optimal cost parallel algorithms for lexicographical ordering. *Purdue University, Tech. Rep.*, 55(2):86–602, 1986. 237

[IMM+03] Costas S. Iliopoulos, Manal Mohamed, Laurent Mouchard, Katerina Perdikuri, William F. Smyth, and Athanasios K. Tsakalidis. String regularities with don't cares. *Nord. J. Comput.*, 10(1):40–51, 2003. 48

[IMR08] Costas S. Iliopoulos, Laurent Mouchard, and M. Sohel Rahman. A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. *Math. in Computer Science*, pages 557–569, 2008. 48

[IS92] Costas S. Iliopoulos and William F. Smyth. Optimal algorithms for computing the canonical form of a circular string. *Theor. Comput. Sci.*, 92(1):87–105, 1992. 134

[ISI+14] Tomohiro I, Shiho Sugimoto, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing palindromic factorizations and palindromic covers on-line. In Alexander S. Kulikov, Sergei O. Kuznetsov, and Pavel A. Pevzner, editors, *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16-18, 2014. Proceedings*, volume 8486 of *Lecture Notes in Computer Science*, pages 150–161. Springer, 2014. 236

[ISM11] Tomohiro I, Inenaga Shunsuke, and Takeda Masayuki. Palindrome pattern matching. In *Proceedings of 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27–29, 2011.*, pages 232 – 245, 2011. 95

[JPB01] Thomas Junier, Marco Pagni, and Philipp Bucher. mmsearch: a motif arrangement language and search program. *Bioinformatics*, 17(12):1234–1235, 2001. 149

[JU91] Petteri Jokinen and Esko Ukkonen. Two algorithms for approximate string matching in static texts. In *Symp. on Mathematical Foundations of Computer Science*, pages 240–248, 1991. 174

[JW04] LB Jorde and SP Wooding. Genetic variation, classification and 'race'. *NATURE GENETICS*, 36(11, Suppl. S):S28–S33, NOV 2004. 171

[KA03] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In Ricardo A. Baeza-Yates, Edgar Ch ávez, and Maxime Crochemore, editors, *Symp. on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2003. 115, 118, 128, 129, 133, 136, 237

[KA05] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *J. Discrete Algorithms*, 3(2–4):143–156, 2005. 42

[KAC14] KACSTAC. King abdulaziz city for science and technology arabic corpus (kacstac), 2014. 200, 222

[KCG90] Mark D. Kernighan, Kenneth W. Church, and William A. Gale. A spelling correction program based on a noisy channel model. In *Proceedings of the 13th conference on Computational linguistics – Volume 2*, COLING '90, pages 205–210, Stroudsburg, PA, USA, 1990. Association for Computational Linguistics. 205, 210

[KDHT98] Viggo Kann, Rickard Domeij, Joachim Hollman, and Mikael Tillenius. Implementation aspects and applications of a spelling correction algorithm, 1998. 205

[Kho01] S. Khoja. Stemming arabic text. *Computing Department, Lancaster University*, pages 68–74, 2001. 195

[KK99] Roman Kolpakov and Gregory Kucherov. On maximal repetitions in words. *J. Discrete Algorithms*, 1:159–186, 1999. 95

[KK09] Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theoretical Computer Science*, pages 5365 – 5373, November 2009. 95, 96

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977. 22, 95, 153

[Knu68] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968. 22

[Knu09] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison–Wesley Professional, 12th edition, 2009. 230

[Knu10] Donald E Knuth. *Selected Papers on Design of Algorithms*. Center for the Study of Language and Information, Stanford, California, 2010. 19

[KR87] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern–matching algorithms, 1987. 178

[KS99] Donald L. Kreher and Douglas R. Stinson. Combinatorial algorithms: generation, enumeration, and search. *SIGACT News*, 30(1):33–35, 1999. 133

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th international conference on Automata, languages and programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer–Verlag. 42

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. 133

[Kuk92] Karen Kukich. Techniques for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, December 1992. 205

[LBC02] Leah S. Larkey, Lisa Ballesteros, and Margaret E. Connell. Improving stemming for arabic information retrieval: light stemming and co–occurrence analysis. *SIGIR*, pages 275–282, 2002. 195

[LBLM11] Estelle Lamprea-Burgunder, Philipp Ludin, and Pascal Mäser. Species–specific typing of DNA based on palindrome frequency patterns. *DNA Research*, 18:117 – 124, 2011. 96

[LD09] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows– Wheeler Transform. *Bioinformatics*, 25(14):1754–1760, July 2009. 106

[Lev66] V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966. 22, 37, 215, 219

[Lip05] R. Lippert. Space–efficient whole genome comparisons with Burrows Wheeler Transforms. *Journal of Computational Biology*, 12(4):407–415, 2005. 19

[Lot83] M. Lothaire. *Combinatorics on Words*. Reading, MA (1983); 2nd Edition, Cambridge University Press, Cambridge (1997). Addison–Wesley, 1983. 106, 117, 119, 134

[Lot05] M. Lothaire, editor. *Applied Combinatorics on Words*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2005. 23, 117, 122

[LS02] Yin Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica, 32–1, 95–106*, 2002. 64, 69

[LSvD+09] Julian Lange, Helen Skaletsky, Saskia K M van Daalen, Stephanie L Embry, Cindy M Korver, Laura G Brown, Robert D Oates, Sherman Silber, Sjoerd Repping, and David C. Page. Isodicentric Y chromosomes and sex disorders as byproducts of homologous recombination that maintains palindromes. *Cell*, 138:855–869, September 2009. 96

[LTPS09] B. Langmead., C. Trapnell., M. Pop, and S. J. Salzberg. Ultrafast and memory–efficient alignment of short DNA sequences to the human genome. *Genome Biol*, 10:R25 – R25, 2009. 106

[LYL+09] R. Li, C. Yu, Y. Li, T. W. Lam, S. M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. 106

[Lyn54] R. C. Lyndon. On burnside problem *i*. *Transactions of the American Mathematical Society*, 77:202 – 215, 1954. 117

[Lyn55] R. C. Lyndon. On burnside problem *ii*. *Transactions of the American Mathematical Society*, 78:329 – 332, 1955. 117

[Man75] Glenn Manacher. A new linear–time on–line algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22:346 – 351, July 1975. 94, 95, 97, 100

[Mar13] A. A. Markov. Essai d'une recherche statistique sur le texte du roman "Eugene Onegin" illustrant la liaison des epreuve en chain (example of astatistical investigation of the text of Eugene Onegin illustrating the dependence between samples in chain). *Izvistia Imperatorskoi Akademii Nauk (Bulletin de l'Académie Impériale des Sciences de St. – Pétersbourg)*, 7:153–162, 1913. 209

[MBY91] Udi Manber and Ricardo A. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Inform. Process. Lett.*, 37(3):133–136, 1991. 22

[McC76] Edward M. McCreight. A space–economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976. 41, 78

[MDM91] Eric Mays, Fred J. Damerau, and Robert L. Mercer. Context based spelling correction. *Inf. Process. Manage.*, 27(5):517–522, September 1991. 206

[Mel96] Guy Melançon. Lyndon factorization of infinite words. In Claude Puech and Rüdiger Reischuk, editors, *Symp. on Theoretical Aspects of Computer Science*, volume 1046 of *Lecture Notes in Computer Science*, pages 147–154. Springer, 1996. 95

[MII+09] Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science*, 410:900–913, March 2009. 95

[ML84] Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms 5*, pages 422–432, 1984. 49

[ML08] Tomás Martínek and Matej Lexa. Hardware acceleration of approximate palindromes searching. pages 65–72, 2008. 95

[MM90] Udi Manber and Gene Myers. Suffix arrays: A new method for on–line string searches. In *Proc. 1st ACM-SIAM Symp. Discrete Algs.*, pages 319–327, 1990. 19, 42

[MM01] Stoyan Mihov and Denis Maurel. Direct construction of minimal acyclic subsequential transducers, 2001. 226

[MNRR13] Tanaeem M. Moosa, Sumaiya Nazeen, M. Sohel Rahman, and Rezwama Reaz. Linear time inference of strings from cover arrays using a binary alphabet. *Disc. Math., Algs. & Appls.*, 2013. 50

[MNSV09] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of individual genomes. In *Proceedings of the 13th Annual International Conference on Research in Computational Molecular Biology*, RECOMB 2'09, pages 121–137, Berlin, Heidelberg, 2009. Springer–Verlag. 171

[MP70] J.H. Morris and Vaughan R. Pratt. A linear pattern–matching algorithm. *Technical Report 40*, (8), 1970. 117, 119

[MPVZ05] Michele Morgante, Alberto Policriti, Nicola Vitacolonna, and Andrea Zuccolo. Structured motifs search. *Journal of Computational Biology*, 12(8):1065–1082, 2005. 148, 150, 157, 158

[MR10] Tanaeem M. Moosa and M. Sohel Rahman. Indexing permutations for binary strings. *Inform. Process. Lett.*, 110(18–19):795–798, September 2010. 83, 84, 85, 235

[MR12] Tanaeem M. Moosa and M. Sohel Rahman. Sub-quadratic time and linear space data structures for permutation matching in binary strings. *J. Discrete Algorithms*, 10:5–9, 2012. 84

[MS94] Dennis Moore and W. F. Smyth. An optimal algorithm to compute all the covers of a string. *Inform. Process. Lett. 50, 239-246*, 1994. 64

[MS95] Dennis Moore and W. F. Smyth. Correction to: An optimal algorithm to compute all the covers of a string. *Inform. Process. Lett. 54 101-103*, 1995. 64

[MS04a] Bruno Martins and Mario J. Silva. Spelling correction for search engine queries. In *In Proceedings of EsTAL–04, Espana for Natural Language Processing*, 2004. 207

[MS04b] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004. 227

[MSM99] Dennis Moore, W. F. Smyth, and Dianne Miller. Counting distinct strings. *Algorithmica 23(1)*, pages 1–13, 1999. 50

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001. 22, 177

[NEGH10] H.M. Noaman, S. Elmougy, A Ghoneim, and T. Hamza. Naive bayes classifier based arabic document categorization. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–5, March 2010. 201

[NM07] Gonzalo Navarro and Veli Mäkinen. Compressed full–text indexes. *ACM Comput. Surv.*, 39(1), 2007. 133

[NR01] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps patternmatching, with application to protein searching. In *RECOMB*, pages 231–240, 2001. 149, 150, 158

[NR03] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, 10(6):903–923, 2003. 149, 150, 158

[NRR12] Sumaiya Nazeen, Sohel M. Rahman, and Rezwana Reaz. Indeterminate string inference algorithms. *J. Discrete Algorithms*, 10:23–34, 2012. 50, 65, 66

[NS13] Shoshana Neuburger and Dina Sokol. Succinct 2D dictionary matching. *Algorithmica*, 65(3):662–684, 2013. 26, 117

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in theamino acid sequence of two proteins. *J. Mol. Biol.*, 48:433–443, 1970. 22

[NZC09] Ge Nong, Sen Zhang, and Wai H. Chan. Linear suffix array construction by almost pure induced–sorting. *Data Compression Conference*, 0:193–202, 2009. 133

[OTAT08] Naoaki Okazaki, Yoshimasa Tsuruoka, Sophia Ananiadou, and Junichi Tsujii. A discriminative candidate generator for string transformations. pages 447–456, 2008. 216

[Par66] Rohit J. Parikh. On context–free languages. *J. Assoc. Comput. Mach.*, 13(4):570–581, 1966. 78, 82

[PB02] Alexandre H L Porto and Valmir C Barbosa. Finding approximate palindromes in strings. *Pattern Recognition*, 2002. 95

[Per05] Ludovic Perret. A chosen ciphertext attack on a public key cryptosystem based on Lyndon words. *IACR Cryptology ePrint Archive*, 2005:14, 2005. 26, 106, 117

[Pin85] R.Y. Pinter. Efficient string matching with dont care patterns. *In A. Apostolico and Z. Galil (Eds.). Combinatorial algorithms on words, NATO Advanced Science Institute Series F: Computer and System Sciences*, 12:11–29, 1985. 22

[PST06] Simon J. Puglisi, William F. Smyth, and Andrew Turpin. Suffix arrays: what are they good for? In *Proceedings of the 17th Australasian Database Conference – Volume 49*, ADC 06, pages 17–18, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc. 42

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), July 2007. 19, 42

[PZ83] J. J. Pollock and A. Zamora. Collection and characterization of spelling errors in scientific and scholarly text. *Journal of the American Society for Information Science*, 34(1):51–58, 1983. 205

[Reu93] C. Reutenauer. *Free Lie algebras*. London Math. Soc. Monographs New Ser. 7. Oxford University Press, 1993. 26, 106, 117

[RH05] Anne Rimrott and Trude Heift. Language learners and generic spell checkers in call – calico journal. 23(1), September 2005. 214

[RIL+06] M. Sohel Rahman, Costas Iliopoulos, Inbok Lee, Manal Mohamed, and W.F. Smyth. Finding patterns with variable length gaps or don't cares. In D.Z. Chen and D.T. Lee, editors, *Annual International Computing & Combinatorics Conference (COCOON)*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2006. 150, 152, 158

[Ris01] Irina Rish. An empirical study of the naive Bayes classifier. In *International Joint Conference on Artificial Intelligence*, pages 41–46, 2001. 201

[RRB+10] C. Anton Rytting, Paul Rodrigues, Tim Buckwalter, David Zajic, Bridget Hirsch, Jeff Carnes, Nathanael Lynn, Sarah Wayland, Chris Taylor,

Jason White, Charles Blake III, Evelyn Browne, Corey Miller, and Tristan Purvis. Error correction for arabic dictionary lookup. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta, may 2010. European Language Resources Association (ELRA). 207

[Rus03] F. Ruskey. *Combinatorial Generation*. (Unpublished book), 1 edition, 2003. 133, 134

[RY98] Eric Sven Ristad and Peter N. Yianilos. Learning string–edit distance. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(5):522–532, 1998. 215, 240

[SAB11] Tengku Mohd T. Sembok, Belal Mustafa Abu Ata, and Zainab Abu Bakar. A rule–based arabic stemming algorithm. In *In Proceedings of the 5th European conference on European computing conference*, pages 392–397, 2011. 194, 195

[SAG03] Khaled Shaalan, Amin Allam, and Abdallah Gomah. Towards automatic spell checking for arabic, 2003. 207

[San72] D. Sankoff. Matching sequences under deletion–insertion constraints. In *Proc. Nat. Acad. Sci. USA 80*, pages 4–6, 1972. 22

[SAP+12] Khaled Shaalan, Mohammed Attia, Pavel Pecina, Younes Samih, and Josef van Genabith. Arabic word generation and modelling for spell checking. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA). 210

[Sav96] Carla Savage. A survey of combinatorial gray codes. *SIAM Review*, 39:605–629, 1996. 134

[SdCC+13] Christian JA Sigrist, Edouard de Castro, Lorenzo Cerutti, Béatrice A Cuche, Nicolas Hulo, Alan Bridge, Lydie Bougueleret, and Ioannis Xenarios. New and continuing developments at prosite. *Nucleic acids research*, 41(D1):D344–D347, 2013. 148, 149, 156

[Sel74] P. H. Sellers. On the theory and computation of evolutionary distance. *SIAM J. Appl. Math.*, 26:787–793, 1974. 22

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell system technical journal*, 27:379–423, 623–656, 1948. 210

[Sha05] Khaled F. Shaalan. Arabic gramcheck: a grammar checker for arabic. *Softw., Pract. Exper.*, 35(7):643–665, 2005. 210

[SK83] David Sankoff and Joseph B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983. 37

[SLLM09] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. A four–stage algorithm for updating a Burrows– Wheeler Transform. *Theoretical Computer Science*, 410(43):4350–4359, October 2009. 106

[Smy03] Bill Smyth. *Computing Patterns in Strings*. Pearson/Addison–Wesley, 2003. 23, 49, 64, 106, 117, 134

[Smy13] W. F. Smyth. Computing regularities in strings: A survey. *Eur. J. Comb.*, 34(1):3–14, 2013. 22, 23

[Sto72] Harold S. Stone. *Introduction to Computer Organization and Data Structures*. McGraw–Hill, New York, 1972 ed. edition, 1972. 19

[Str13] Stringology.org. Stringology, 2013. 20

[SVM11] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes. In Teresa M. Przytycka and Marie-France Sagot, editors, *WABI*, volume 6833 of *Lecture Notes in Computer Science*, pages 270–281. Springer, 2011. 133

[SW08] W. F. Smyth and Shu Wang. New perspectives on the prefix array. *Proc. 15th String Processing & Inform. Retrieval Symp. (SPIRE)*, 5280:133–143, 2008. 49, 65, 66

[SW09a] W. F. Smyth and Shu Wang. An adaptive hybrid pattern-matching algorithm on indeterminate strings. *Internat. J. Foundations of Computer Science*, 20(6):985–1004, 2009. 65

[SW09b] W. F. Smyth and Shu Wang. A new approach to the periodicity lemma on strings with holes. *Theoret. Comput. Sci.*, 410(43):4295 – 4302, 2009. 65

[TBYT06] Hisashi Tanaka, Donald A Bergstrom, Meng-Chao Yao, and Stephen J Tapscott. Large DNA palindromes as a common form of structural chromosome aberrations in human cancers. *Human Cell*, 19(1):17–23, 2006. 96

[TEC05] Kazem Taghva, Rania Elkhoury, and Jeffrey S. Coombs. Arabic stemming without a root dictionary. pages 152–157. IEEE Computer Society, 2005. 195

[Thu06] Axel Thue. über unendliche zeichenreihen. *Norske Vid. Selsk. Skr. Mat. Nat. Kl. (Cristiania)*, (7):1–22, 1906. 22

[TM02] Kristina Toutanova and Robert C. Moore. Pronunciation modeling for improved spelling correction. pages 144–151. ACL, 2002. 205, 215

[Ukk85] Esko Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985. 22

[Ukk92] Esko Ukkonen. Approximate string–matching with q–grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, January 1992. 22, 173, 208

[Ukk95] Esko Ukkonen. On–line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. 41, 42, 78

[UW93] Esko Ukkonen and Derick Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993. 174

[VM05] M. Voráček and B. Melichar. Searching for regularities in generalized strings using finite automata. *Proc. Internat. Conf. on Numerical Analysis & Applied Maths. Wiley-VCH*, 2005. 65

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973. 41, 78

[Wik14] Wikipedia. String (computer science), 2014. 21

[ZA03] Chiraz Ben Othmane Zribi and Mohamed Ben Ahmed. Efficient automatic correction of misspelled arabic words based on contextual information. volume 2773 of *Lecture Notes in Computer Science*, pages 770–777. Springer, 2003. 209

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977. 95

[ZPZ81] E. M. Zamora, J. J. Pollock, and Antonio Zamora. The use of trigram analysis for spelling error detection. In *Information Processing and Management*, pages 305–316, 1981. 206

[ZZ06] Yongqiang Zhang and Mohammed J. Zaki. SMOTIF: efficient structured pattern and profile motif search. *Algorithms for Molecular Biology*, 1, 2006. 150, 159